V Facoltà di Ingegneria
Politecnico di Milano

Portable DBMS: analysis, specification and prototype implementation of a JDBC™ layer

Tesi di Laurea di

Federico Maggi
matr. 653802

Relatore Chiar.ma Prof.ssa C. Bolchini

Correlatore Chiar.ma Prof.ssa L. Tanca

Laurea Ingegneria Informatica
Anno Accademico 2003-2004

V Facoltà di Ingegneria
Politecnico di Milano

Portable DBMS: analysis, specification and prototype implementation of a JDBC™ layer

Tesi di Laurea di

Federico Maggi
matr. 653802

Relatore Chiar.ma Prof.ssa C. Bolchini

Correlatore Chiar.ma Prof.ssa L. Tanca

Laurea Ingegneria Informatica
Anno Accademico 2003-2004

Acknowledgements

É solo apparentemente semplice scrivere un paragrafo come questo: chi citare per primo? Chi per ultimo? Ma soprattutto...*chi* includere nella lista? E quale ordine seguire? Non è facile! Non voglio dimenticare nessuno e mi scuso in anticipo se questo dovesse accadere al termine di queste righe. Bene, i preliminari ci sono, ora cominciamo con i ringraziamenti, quelli *veri*.

Sono molte le persone che, più o meno direttamente/inconsciamente – hanno contribuito alla realizzazione di questo lavoro. La persona più diretta in assoluto è senza dubbio la Prof.ssa Cristiana Bolchini, che ringrazio di cuore: tanto per avermi dato la possibilità di entrare a far parte del *team di sviluppo*, tanto per le pazienti correzioni, gli aggiustamenti e i consigli. Allo stesso modo, un sincero ringraziamento va alla Prof.ssa Letizia Tanca perchè con quel suo "...Maggi metterebbe entità deboli dappertutto..." mi ha suggerito che forse un E-R va guardato più di una volta!
Beh, parlando del *team di sviluppo*, un grazie più *leet* degli altri va a Krl e George senza i quali (1) questo driver dove lo attaccavo? (2) PoLiDBMS sarebbe stato davvero duro da *hacckare*! Grazie per le vostre mail, sempre puntuali e precise nelle risposte.
(Prima di uscire dal poli mancano ancora molte persone) Paola! Senza la quale l'algebra e le macchine di Turing sarebbero state un incomprensibile guazzabuglio! Un sincero grazie anche al Prof. Dino Mandrioli perchè con il suo "...altrimenti potete usare una macchina di Turing, che tanto è gratis!" mi ha fatto capire che è vero che "...servono 50 minuti per pensare e 10 minuti per scrivere...".
Ricordando i pomeriggi passati davanti all'eserciziario di IT, ringrazio Simone senza il quale probabilmente anche altri esami sarebbero stati decisamente più

duri! Allo stesso modo ringrazio Paola *Casalina* e Marco *Gatto* perchè dal giorno di "`IntSet > Intset > Insett > Insetto`" ad oggi sono sempre stati ottimi amici e compagni di studio all'ISU. Già che siamo vicini all'ISU, non posso certo dimenticare i preziosi consigli di Giovanni, che mi ha fatto capire con pazienza gli amplificatori.

Un grazie anche a Schumy, Nico e tutta la *crew*, senza i quali non avrei mai avuto la forza di volontà di affrontare gli esercizi di segnali, o di automatica...

Un grazie davvero *geek* va indubbiamente a *mij* a *oettam*, *m3ur1* e Mirjana, Roberta, Andrea, Anna Maria, Carmen (chi avrebbe sopportato i miei periodi neri pre-esame?) e tutti quelli che hanno a che fare con questo fantastico gruppo di compagni di studio. Grazie anche a Luca e Giorgio, compagni di progetto, e all'Ing. Stefano Modafferi per i suoi consigli e le critiche sempre costruttive.

Insomma, tutti coloro grazie ai quali non ho mai mollato, grazie ai quali la mia vita al poli è stata migliore...li ringrazio, nessuno escluso. Ringrazio anche chi la vita me l'ha resa difficile, altrimenti che gusto ci sarebbe stato?!

Bene, finito con il poli, cominciamo con i saluti rivolti a chi fa parte della mia vita nell'hinterland. Ringrazio di cuore i miei genitori, Adriana ed Eraldo, che mi hanno sempre posto di fronte più di un'alternativa ad ogni mio pensiero, che mi hanno sempre supportato e sOpportato, che hanno sempre assecondato le mie scelte, che in tutti questi anni non mi hanno mai privato di niente, che mi hanno fatto imparare il giusto peso da dare alle cose...

Ringrazio Alice, sia per quello che rappresenta per me, sia perchè senza di lei il TOEFL avrebbe dovuto aspettare ma soprattutto perchè senza di lei continuerei a vivere soltanto in questo mondo.

Un grandissimo saluto va sicuramente anche a tutta la compagnia di Ornago, per aver sopportato tutte quelle fotografie e tutti i miei torpiloqui mentre raccontavo di questo lavoro: grazie Edo e Gemello, Wally, Sele, Vale, Villaz, Gigio, Steve, Ele e tutti quanti!

Tra gli amici non posso assolutamente dimenticare Tommy, Faz e ValeVale: grazie

per la vostra sincera amicizia!
Sembra incredibile ma senza tutte queste persone, e tante altre che non sono qui scritte ma che porto giornalmente nel cuore, questo lavoro non avrebbe preso la forma che ha.
In ultimo, ma non per questo meno importante, ringrazio Giò Small, per la sua spontaneità, ed Elli per esserci sempre stata.

Federico

Dedicato

ai miei familiari.

In memoria

di mia nonna Rosetta.

“You cannot not *communicate”*

P. Watzlavick

PREFACE

Bill “Ciao Steve! Come va?”

Steve “Sono un po’ nervoso oggi: abbiamo qualche problema con il nostro ultimo progetto.”

Bill “Progetto? Quale progetto?”

Steve “Abbiamo appena finito un programma tipo organizer per un PDA. Il problema è che non riusciamo a connettere il database sottostante.”

Bill “Che DBMS usate? Un DBMS lite suppongo...”

Steve “Sì esatto, è PoLiDBMS. É un Very Small DBMS scritto in Java. Il fatto è non ha una API standard. Il mio progetto è spacciato!”

Bill “Probabilmente no. Dà un’occhiata a questa tesi, arriva dal Politecnico di Milano e sembra fare al caso tuo. L’autore ha sviluppato PoLiJDBC...”

Steve “PoLiJDBC? Il nome suona molto bene. Ma cos’è? Un driver JDBC o qualcosa di simile?”

Bill “Esatto! É un Very Small JDBC driver, per essere precisi.”

Steve “Bene, allora devo proprio leggermi la documentazione.”

Hai sviluppato un'applicazione portabile? Hai bisogno di manipolare piccoli frammenti di dati? La tua applicazione è scritta in Java? Noi abbiamo quel che fa al caso tuo! Il primo prototipo di 'small-but-powerful JDBC™ driver' è stato rilasciato...e funziona! É PoLiJDBC e il nome sta per Portable Little JDBC. Questo nuovo driver è stato creato per 'aprire le porte' a PoLiDBMS, un DBMS portabile e completo privo però di una API standardizzata.

L'importanza di avere una API standard è più che ovvia; tutti i maggiori DBMS commerciali come Oracle 10g[1] o DB2 Everyplace[2] di IBM e anche prodotti open-source come HyperSonic SLQ[3], tinySQL[4], Derby Project[5] di Apache, etc, hanno tutti un driver JDBC™. Al contrario, un ottimo DBMS che però sia privo di una buona API è meno di niente! Ma ora anche PoLiDBMS può contare sul suo nuovo e potente driver: PoLiJDBC!

Con la sua architettura modulare, PoLiJDBC è perfetto per PoLiDBMS; è leggero, supporta le transazioni locali, è estendibile ed è stato scritto da zero, seguendo il cosiddetto *approccio di scalabilità* ([5]). Questo driver di nuova generazione non soltanto mette a disposizione l'API standard specificata da JDBC™ ma enfatizza anche le peculiarità di PoLiDBMS. L'interfaccia standard è stata estesa in accordo con PoLiDBMS in modo tale da supportare pienamente le sue particolari funzionalità; in questo modo le applicazioni esistenti potranno essere tanto 'compatibile con JDBC' quanto 'compatibile con PoLiDBMS', senza necessità di costose modifiche nel codice. Il nostro lavoro è stato fortemente motivato dall'idea di "offrire qualcosa di davvero utile sia per gli sviluppatori futuri che per gli sviluppatori di applicazioni".

Come suggerisce la Figura 1, possiamo collocare PoLiJDBC nell'area di ricerca VSDB, un progetto di ampio respiro nato due anni fa e che sviluppa metodologie

[1]`http://www.oracle.com/technology/documentation/database10g.html`
[2]`http://www-306.ibm.com/software/data/db2/everyplace/`
[3]`http://hsqldb.sourceforge.net/`
[4]`http://www.jepstone.net/tinySQL/`
[5]`http://incubator.apache.org/derby/`

e tecniche per affrontare il design di applicazioni ‘leggere‘ per la manipolazione dati. Con lo scopo principale di memorizzare informazioni di dimensione relativamente ristretta, una base di dati “molto piccola” trova impiego in applicazioni portabili come (micro) sistemi informativi personali, database personali di viaggio o finanziari e cartelle cliniche personali. Il crescente utilizzo di accessori come computer palmari, telefoni cellulari e Smart Card ha senza dubbio dato una forte spinta allo sviluppo di talune applicazioni, è stato pertanto necessario un approccio metodologico, pur considerando che le recenti tecnologie producono dispositivi piccoli ma comunque potenti e capienti.

I ricercatori del progetto VSDB raccomandano il cosiddetto *approccio di scalabilità*[6], proposto per la prima volta da Bobineau in [5], invece di adattare le esistenti applicazioni per la gestione dei dati al mondo “in piccolo”; la metodologia guida il progettista di basi di dati “scalando” le tecnologie, le tecniche e le conoscenze attuali alle limitazioni del dispositivo.

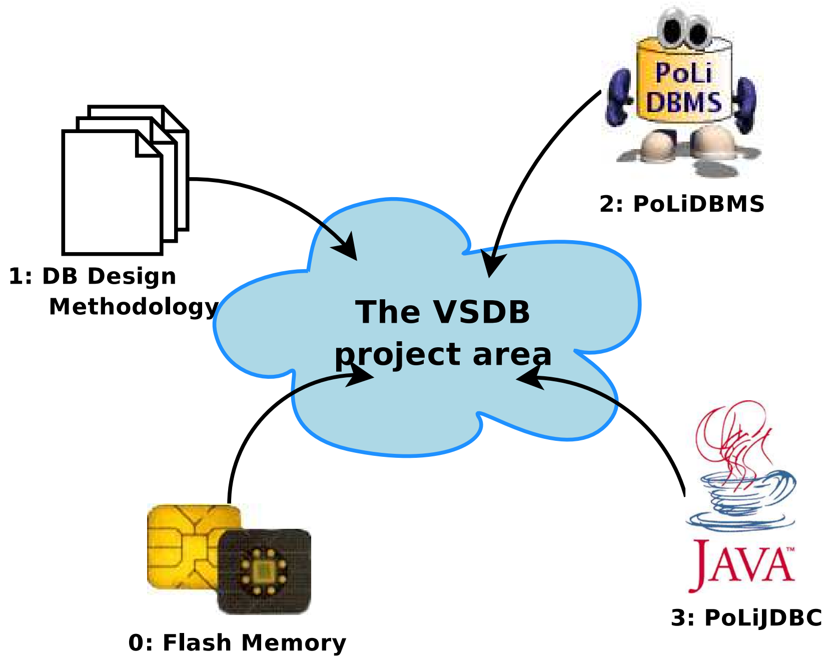


Figure 1: PoLiJDBC come parte del progetto VSDB.

Come “PoLiDBMS è nato per essere il DBMS del progetto VSDB”, allo stesso

[6] *“scaling down approach”*

modo “PoLiJDBC è il primo prototipo di driver JDBC™ per il progetto VSDB”. Se “PoLiDBMS è un Very Small DBMS”, possiamo classificare PoLiJDBC come un Very Small JDBC™ driver; un driver per il quale l’enorme tecnologia JDBC™ è stata *smussata*, *ripulita* e *rimpicciolita*. Sia le limitazioni software che quelle dovute all’hardware sono state prese in considerazione sin dai primi passi; innanzitutto abbiamo visionato la tecnologia JDBC™ considerando tanto le paculiarità di PoLiDBMS, tanto il sistema, con l’aiuto delle metodologie sopraccitate ([5], [6], [7]) e del caso di studio sul design di VSDB condotto in precedenza, [14]. Durante questa prima macro-fase “JDBC™ ha incontrato PoLiDBMS” per fusione e intersezione della tecnologia JDBC™ con le caratteristiche di PoLiDBMS. La Figura 2 mostra l’architettura di alto livello di PoLiDBMS mettendo in evidenza l’area di interesse per questo progetto, area sulla quale abbiamo condotto i nostri primi studi.

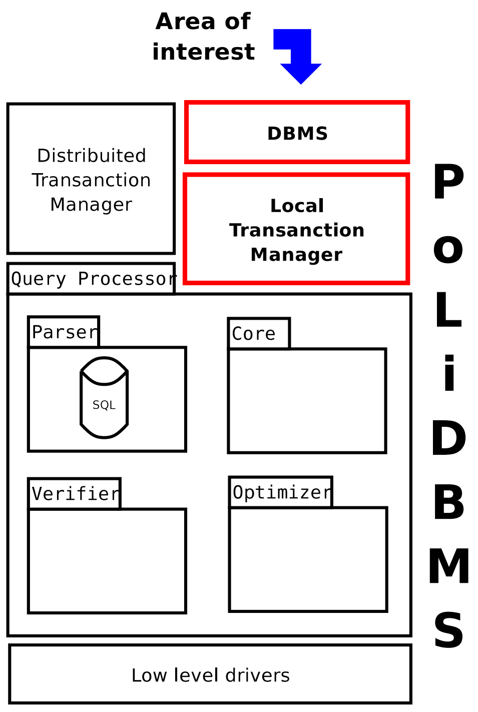


Figure 2: Gli strati di alto livello di PoLiDBMS. L’area di interesse per questo progetto è stata evidenziata.

Cosiccome è stato fatto dalla letteratura esistente, abbiamo optato per un approccio procedurale e metodologico per “scalare” l’API JDBC™: come risultato, è stato creato una bozza di 'workflow' che ci ha guidati nei principali passi successivi: *selezione delle funzioni*, *specifiche*, *scrittura degli algoritmi*, *selezione dei tipi di dato* da supportare e la scelta dei *meccanismi per future estensioni*. Anche la complessità spaziale e temporale degli algoritmi utilizzati è presa in considerazione da tale procedura, in modo da alleggerire il più possibile il carico di lavoro della CPU e da limitare la memoria di lavoro necessaria.

L'idea che accompagna tutto il progetto è che PoLiJDBC non debba essere considerato semplicemente un driver JDBC™; come già accennato, PoLiDBMS ha delle particolari caratteristiche e il nostro *scopo* è quello di *mantenere ogni minimo dettaglio* mettendo a disposizione eventuali interfacce e metodi extra. Per questo motivo, sono state create anche due tipologie di manuali utente: un manuale per l'utilizzo di base del driver e un secondo e più avanzato manuale per sviluppatori che include la documentazione del codice sorgente e delle funzionalità avanzate, [10].

Come tutti i processi di sviluppo software, il nostro lavoro ha seguito una versione adattata del classico processo di sviluppo ciclico; basandosi sull'ordinario 'workflow', il lavoro è stato suddiviso in un certo numero di *fasi* (chiamate *attività*, per conformità alla terminologia UML) come suggerisce Liskov in [4]. La Figura 3 tratteggia tutte le macro-attività del progetto; alcune attività del caso sono state aggiunte allo schema classico:
Documentazione funzionalità non supportate e tutte le attività preliminari.

Dal momento che PoLiJDBC è parte dello sviluppo di PoLiDBMS, abbiamo fatto affidamento sull'esistente infrastruttura di gestione del progetto; in questo modo tutte le interazioni, le procedure e i rapporti sullo stato di sviluppo si sono rivelate semplici, grazie all'automazione offerta dai tool di sviluppo e dai repository.

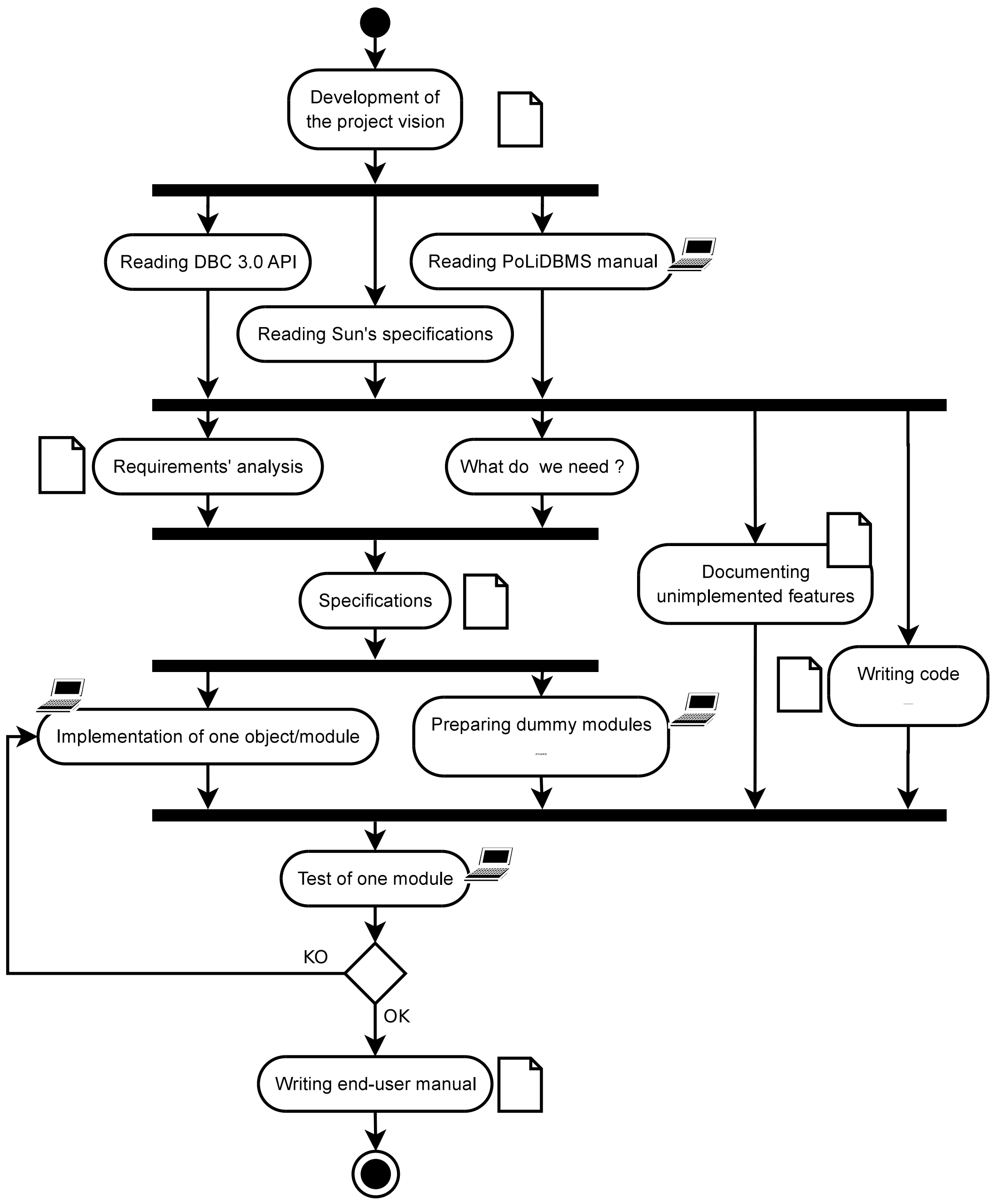


Figure 3: Un activity-diagram che mostra tutte le macro-attività del progetto.

Contents

List of Tables

List of Figures

Chapter I

Introduction

Bill “Hi Steve! How are you?”

Steve “I’m a little bit nervous today: we got quite a few problems with our last project.”

Bill “Project? What project?”

Steve “We’ve just finished a PIM application to be run on a Java enabled PDA. The problem is that we can’t figure out how to connect the database.”

Bill “Which DBMS do you use? A small one, I guess...”

Steve “You guess right; it’s PoLiDBMS, a very small DBMS written in Java. The problem is that it doesn’t have a standard API! Our project is lost!”

Bill “May be not. Look at this thesis, it comes from Politecnico di Milano, and it seems to be what you’re looking for; the author has developed PoLiJDBC...”

Steve “PoLiJDBC? That name sounds good for me. What’s that? Is it a JDBC driver or something like that?”

Bill “Yeah! It’s a Very Small JDBC™ driver, to be more precise.”

Steve “Well, I must read the documentation right away.”

Is your application a portable one? Do you need to manage small data? Is your application a pure Java one? We have got what you need! The first small-but-powerful JDBC™ driver prototype is out...and it works! Its name is PoLiJDBC, which stands for Portable Little JDBC™ . This fresh driver will 'open the door' for PoLiDBMS, a portable DBMS with a complete feature set but without a standard API.

The importance of a standard API is more than obvious; all commercial DBMSes such as Oracle 10^{g}[1] or IBM's DB2 Everyplace[2] and open-source ones like HyperSonic SQL[3], tinySQL[4], Apache's Derby Project[5], etc, they all have a JDBC™ driver. On the other hand, a great DBMS without a good standard API is less than nothing! But now, PoLiDBMS can reckon its new and powerful driver: PoLiJDBC!

With its highly modular architecture, PoLiJDBC perfectly fits PoLiDBMS' environment; it is small, it supports local transactions, it is extensible and has been written from scratch following a so called *scaling down approach* ([5]). This new generation driver does not only provide the standard JDBC™ APIs but it enforces PoLiDBMS peculiarities. The standard API has been extended w.r.t. PoLiDBMS so its particular features – such as transaction boundaries, data types, etc – are fully supported and existent applications can be both JDBC™ compliant and PoLiDBMS compliant, without expensive code modifications. Our work has been strongly motivated by the idea of "offering something of very useful for both future developers and application developers".

As Figure 4 suggests, we can introduce PoLiJDBC in the VSDB research, a two years old wide-spectrum project that provides methodologies and techniques to approach the development of *small* data manipulation applications. Small databases' main purpose is to hold limited size information, so they are commonly used in various 'portable application' such as *personal (micro) information systems*, *personal medical records*, *personal financial databases* or *personal travel databases*. The growing use of portable devices like Personal Data Assistants,

[1]`http://www.oracle.com/technology/documentation/database10g.html`
[2]`http://www-306.ibm.com/software/data/db2/everyplace/`
[3]`http://hsqldb.sourceforge.net/`
[4]`http://www.jepstone.net/tinySQL/`
[5]`http://incubator.apache.org/derby/`

PDAs, Smart Cards, PalmPCs and Handy Phones has certainly enhanced the development of such portable applications so a methodological approach became strongly required, even if considering that new devices are powerful and with no strict memory limitations.

VSDB's researchers recommend the so called *scaling down approach*, first proposed by Bobineau in [5] rather than adapting an existing data management application for the new *small* environment, their methodology aims at guiding the database design process at different levels (conceptual, logical, physical) by "scaling down" ordinary database techniques "so they perform well under" device's "limitations".

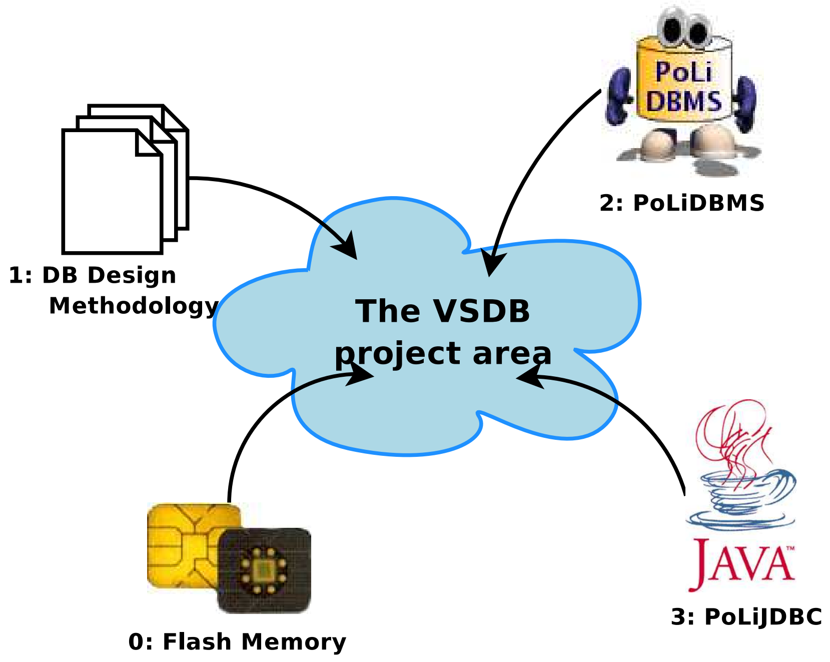


Figure 4: PoLiJDBC as a part of the VSDB project.

As PoLiDBMS "is born as the DataBase Management System for the VSDB project" ([8]), PoLiJDBC "is the first prototype of a JDBC™ driver for the VSDB project". If "PoLiDBMS is a Very Small DBMS", we could classify PoLiJDBC as a Very Small JDBC™ driver; a driver for which the 'huge' JDBC™ technology has been *refined*, *cleaned* and *made smaller*.
Both software and hardware limitations have been taken into account from the very beginning; we first over viewed the JDBC™ technology w.r.t. PoLiDBMS peculiarities and analyzed the particular system environment with the help of both the above mentioned VSDB methodologies ([5], [6], [7]) and the VSDB design case

study we worked on, [14].

During this first macro-step "JDBC™ has met PoLiDBMS" by merging and intersecting JDBC™ technology with PoLiDBMS' features and behaviors. Figure 5 shows an high-level view of PoLiDBMS and highlights which is the *area of interest* for the project; on this area, we conducted our first studies about the DBMS.

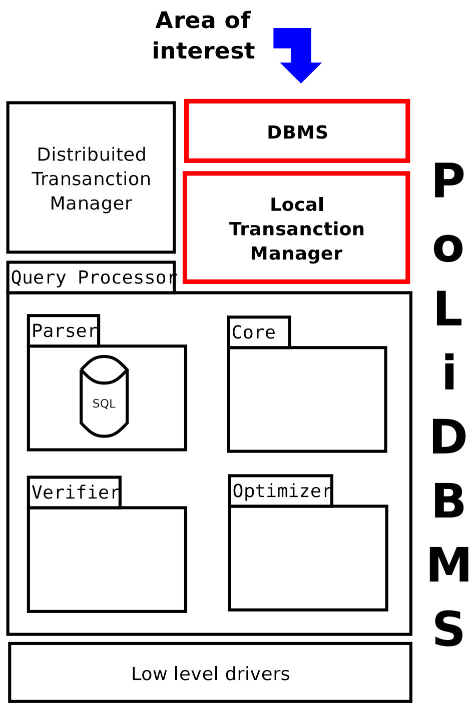


Figure 5: The PoLiDBMS high-level structure. This project's area of interest is highlighted.

According to the existing literature, we opted for a procedural and methodological approach for coping with the *scaling down problem*: as a result, we coined out a workflow draft that has guided us during next steps such as *feature selection*, *design specifications*, *algorithm design*, *data types support* and *future extension mechanisms*. The algorithm temporal and spatial complexity is also regarded by the produced workflow, so both CPU and RAM are only weakly stressed.

The common idea of all the project is that PoLiJDBC is not meant to be only a plain JDBC™ driver; as we mentioned above, PoLiDBMS has very special characteristics and our *goal* is to *maintain any minimal detail* providing extra methods and interfaces. For this reason, two different kinds of documentation have been written; a basic *user manual* for a common usage, and a more advanced *developer* *manual* on which advanced functionalities and code documentation have been reported, [10].

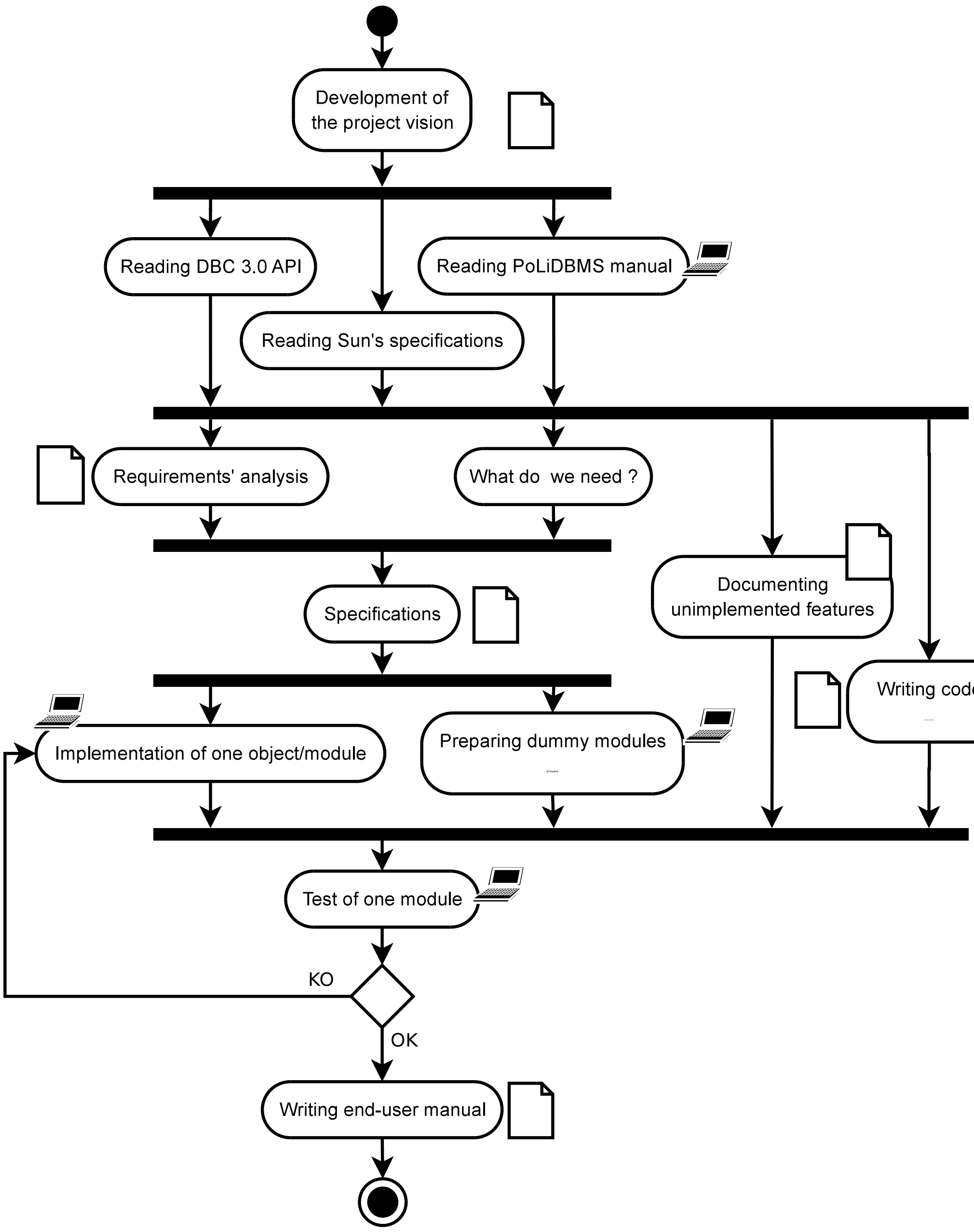


Figure 6: An activity-diagram showing all the project macro-activities/steps.

As a software development project, our work has followed an adapted version of the classical *software life cycle*; starting from the usual workflow, we broken up our development process into a number of *phases* (that we call *activities*, to follow a UML approach) as Liskov suggests in [4]. Figure 6 depicts all the project macro-activities; custom activities like all the *preliminary phases* and the

$\boxed{\text{Documenting unimplemented features}}$ activity were included in the schedule.

Since PoLiJDBC is part of PoLiDBMS' development, we joined the existing project infrastructure; all the project interactions, procedures and status reporting were extremely easy thanks to the automation offered by development tools and repositories.

This document's structure reflects the logical path that Figure 6 shows, and is organized as follows. Chapter 2 discloses the project vision and the *scaling down procedure* is discussed; this chapter also includes a brief overview of the JDBC™ technology as a whole. The next chapter (Chapter 3) covers requirements' analysis and design specifications where our studies are detailed in a classical form. Chapter 4 has been written after the implementation step and it is in charge of reporting the results; together with design specifications, this part should give a complete and comprehensive view of the driver status. After several testing sessions, Section 4.2 has been added. Before dealing with PoLiJDBC on top of PoLiDBMS, the reading of the Appendix is strongly recommended; the user manual and a brief glossary is here included.

Chapter II

Project vision

This chapter illustrates our vision of the project. The first two sections provide a brief overview on Sun® JDBC™ Technology; what JDBC™ is, what role it has in distributed systems, how it is commonly used and eventually how it should be adapted to best fit on small DBMSes. For this first part, major references are the "JDBC™ API Tutorial and Reference, Third Edition" [15] and "JDBC™ 3.0 API Specifications" [13]. The presentation then investigates our expectations from a JDBC™ layer; focusing on our main needs and how to to use this *specialized middleware* for applications. References for this issue are "A methodology for Very Small DataBase Design" [6] and "Logical and Physical Design Issues for Smart Card Databases" [7].

2.1 Sun® JDBC™ 3.0 overview

In this section we propose a brief overview of the Sun®'s JDBC™ Technology, to explore the standard, the specifications and Sun® suggestions to driver developers. Since the most used and the most complete JDBC™ Application Program Interface (API) is the 3rd release, this version has been chosen to realize the driver in order to have a large feature set to select from.

JDBC™ API is a Java API to provide programmatic access to virtually any kind of data (where *data* closely means *organized data*), especially relational data [13, 15]. It was introduced in January 1997 and it quickly become a widely adopted

API. It can be seen as an abstract view of a (relational) database; through its classes and its well specified interfaces, it makes it easy to connect to a data source and retrieve queries' results. More precisely it allows the following operations:

1. **Establish a connection with a DBMS (or any other kind of data source)**
2. **Send SQL statements (`INSERT`, `UPDATE`, `DELETE`, ...) to the data source**
3. **Retrieve and process the results**

The JDBC™ API can fit on different system; thanks to its flexibility, it can easy be positioned in a two or three tier architecture (Figure 7) and can be set on top of a wide variety of data sources including common DBMSes, legacy file systems, tabular text files and object-oriented systems.

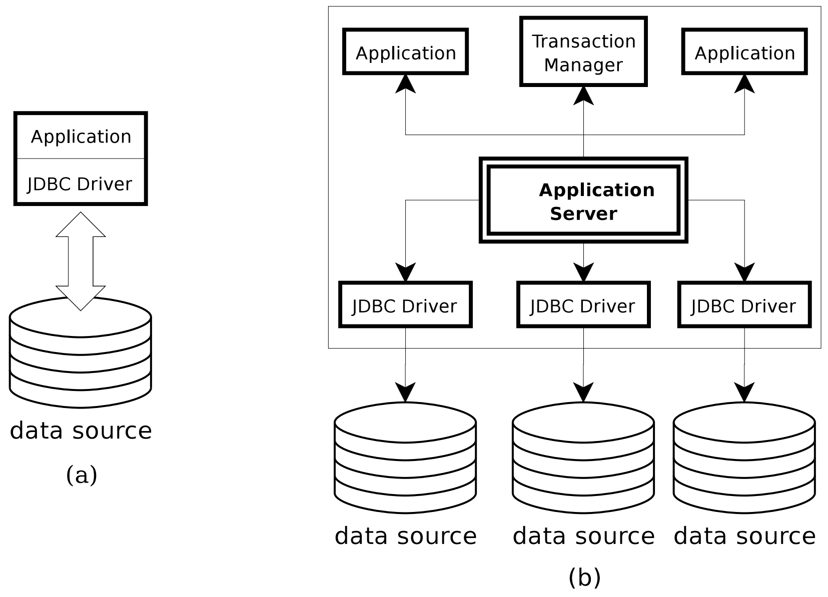


Figure 7: JDBC™ in a (a) two-tier and in a (b) three-tier model

2.1.1 Goals & Specifications

In its first release the JDBC™ API has been thought to provide a thin layer to build a call-level abstraction over SQL databases. At this time the JDBC™ API goes beyond its initial purpose; it is a mature technology that also includes some advanced features for high-end applications like multiple-tier distributed systems.

JDBC™ 3.0 API main goals are enumerated in the following list.

1. **Fit into both J2EE and J2SE platforms**
2. **Be consistent with SQL99 and maintain the focus on SQL**
3. **Consolidate predecessor specifications and maintain backward compatibility with existing applications and drivers**
4. **Keep it simple by offering vendor-neutral access to data sources**
5. **Be reliable, be available and be scalable ("Write once, run everywhere™")**
6. **Specify requirements unambiguously**

The JDBC™ API is based on the X/Open SQL Call Level Interface[1], which is also the basis for Open DataBase Connectivity (ODBC). JDBC™ provides a natural and easy-to-use mapping from the Java programming language to the abstractions defined in the X/Open CLI and SQL standards ([13]).

The most important and significant parts of the API specifications are here presented by following the enumeration used in the previous operations list: each point is clarified by means of sample code.

1. Establishing a connection — The JDBC™ API defines the `Connection` interface to represent a connection (session) to a specific data source. There are two mechanisms to establish a new connection, using two different interfaces/classes

`DriverManager` — it is a fully implemented class that manages a set of drivers. Each specific driver can be loaded according to the specific data source

[1]The X/Open SQL Call Level Interface (SQL CLI) provides an alternative method for applications to access databases than via embedded SQL. It provides a library of functions that can be linked-in and called at run-time to enable any application, such as spreadsheets, word-processors, diary managers, etc., to make use of, and update if necessary, the data stored in databases. (Source: `http://www.xopen.co.uk/`).

(Example; `PGDriver` for PostgreSQL, `MYDriver` for MySQL, etc.). When its `getConnection` method is invoked, a new `Connection` is instantiated and returned back to the caller. This class was included in JDBC™ 1.0 API.

`DataSource` — this is an interface introduced in the JDBC™ 2.0 API, preferred with respect to `DriverManager` because of several reasons

1. it allows the explicitation of the specific underlying data source
2. it transparently *represents* the data source
3. it only requires a change in its parameters to switch from a (real) data source to another, no different objects are needed
4. it supports the `ConnectionPoolDataSource` extension (that caches and reuses `Connection` objects for performance improvement) and the `XA-DataSource` extension (generating `Connection` objects that can participate in distributed transactions)

The following sample code illustrates how to connect to a data source using the two different presented methods.

The use of `DriverManager` is more straightforward and it consist of one phase that is entirely done by the end-user.

```
/*
 * Locate and load the Driver; this automatically calls the
 * DriverManager.registerDriver() method to make foo.bar.db.Driver
 * available.
 */
Class.forName("foo.bar.db.Driver");

/*
 * Get a connection from the first driver in DriverManager
 */
Connection con = DriverManager.getConnection(
                                "jdbc:FooBarDbURL",
                                "myId", "myPwd");
```

The use of DataSource is more complex because it includes a deployment phase that is done by a system administrator; the next phase implies the lookup of the DataSource object through a naming directory (like the Java Naming Directory Interface, JNDI) and the get-connection phase is performed by end-user and the returned Connection object is the same one as the one returned by a DriverManager.

```
/*
 * Deployment phase (JNDI)
 */
Context ctx = new InitialContext();

FooDataSource ds = new FooDataSource();
ds.setServerName("fooServer");
ds.setDataBaseName("fooDb");
ctx.bind("fooDataSource", ds);

/*
 * Lookup and get-connection phase
 */
DataSource ds = (DataSource)ctx.lookup("fooDb");
Connection con = ds.getConnection("myId", "myPwd");
```

2. Send SQL statements Given a Connection instance, some Statements objects can be created from that object to send SQL statements. JDBC™ makes use of three types of statements, that are

Statement — that represents a basic SQL statement;

PreparedStatement extends Statement — that represents a prepared (precompiled) SQL statement; it is like a templated-statement on which some parameters have to be filled before retrieving results;

CallableStatement extends PreparedStatement — it is used for stored procedures: it uses the same templating fashion of PreparedStatement and they are quite similar.

The following sample code illustrates the behaviors of three interfaces (they assume that an existing connection, `con`, is available).

```
/*
 * Create a new Statement
 */
Statement stm = con.createStatement();

/*
 * Send a specific SQL statement
 */
ResultSet rs = stm.executeQuery("SELECT * FROM foo");
```

```
/*
 * Create a new PreparedStatement
 */
PreparedStatement ps = con.prepareStatement(
                  "SELECT * FROM foo WHERE bar = ?");

/*
 * Fill in the ‘bar‘ parameter
 */
ps.setString(1, "barValue"); //bar
```

```
/*
 * Create a new CallableStatement
 */
CallableStatement cs = con.prepareCall(
                     "{CALL PROC(?, "SecondValue", ?)}");

/*
 * Fill in the ‘first‘ and the ‘third‘ paramenter
 */
cs.setString(1, "FirstValue");
cs.setString(2, "ThirdValue");
```

3. Retrieve and process the results The main interface for result retrieval is `ResultSet`, that represents a set of tabular records that can be browsed with one or more row-cursors. There are different kinds of `ResultSet` with peculiar characteristics and functionality; using three parameters, the `ResultSet`’s style can be changed. Such parameters are the following ones

Cursor — modes in which the cursor can be moved across the whole set:

- `TYPE_FORWARD_ONLY` — the set of results cannot be scrolled and the rows are accessed from the first to the last one;
- `TYPE_SCROLL_INSENSITIVE` — the set of results is scrollable both forward and backwards but it is not sensible to changes made on real data source while the result set is open. The cursor can be moved on absolute positions;
- `TYPE_SCROLL_SENSITIVE` — similar to `TYPE_SCROLL_INSENSITIVE` but sensible to variations made on real data source while the object is open.

Concurrency — supported update functionality levels:

- `CONCUR_READ_ONLY` — no changes are allowed;
- `CONCUR_UPDATABLE` — updates can be performed using `ResultSet`'s methods.

Holdability — in a transactional environment (see the next paragraph) this parameter specifying how the cursor will be treated after a transaction commitment:

- `HOLD_CURSORS_OVER_COMMIT` — all cursors are held after any commit operation;
- `CLOSE_CURSORS_AFTER_COMMIT` — all cursors are cleared after any commit operation.

The above parameters can be specified using some dedicated constant fields provided by the `ResultSet` interface definition; Table 1 summarizes such constants for each parameter.

The following sample code shows how different kinds of `ResultSet`s can be created and used for results manipulation. An existent connection object, `con`, is assumed.

Parameter	**Constant Field**	**Default**
TYPE_FORWARD_ONLY	ResultSet.TYPE_FORWARD_ONLY	×
TYPE_SCROLL_SENSITIVE	ResultSet.TYPE_SCROLL_SENSITIVE	
TYPE_SCROLL_INSENSITIVE	ResultSet.TYPE_SCROLL_INSENSITIVE	
CONCUR_READ_ONLY	ResultSet.CONCUR_READ_ONLY	×
CONCUR_UPDATABLE	ResultSet.CONCUR_UPDATABLE	
HOLD_CURSORS_OVER_COMMIT	ResultSet.HOLD_CURSORS_OVER_COMMIT	
CLOSE_CURSORS_AFTER_COMMIT	ResultSet.CLOSE_CURSORS_AFTER_COMMIT	

Table 1: Summary for `ResultSet`'s constant fields. The default value for holdability is implementation defined

```
/*
 * Default values are used
 */
Statement stm = con.createStatement();
ResultSet rs = stm.executeQuery("SELECT * FROM foo");


while (rs.next()) {
    ...
}
```

```
/*
 * Parameter setting is done by an overloaded method
 */
Statement stm1 = con.createStatement(
                       ResultSet.TYPE_SCROLL_SENSITIVE);

Statement stm2 = con.createStatement(
                       ResultSet.TYPE_SCROLL_SENSITIVE,
       ResultSet.CONCUR_UPDATABLE);

Statement stm3 = con.createStatement(
                       ResultSet.TYPE_SCROLL_SENSITIVE,
       ResultSet.CONCUR_UPDATABLE,
       ResultSet.HOLD_CURSORS_OVER_COMMIT);

ResultSet rs1 = stm1.executeQuery("SELECT * FROM foo");
rs1.absolute(3); //move cursor at the 3rd absolute position

ResultSet rs2 = stm2.executeQuery("SELECT * FROM foo");
rs2.first();
rs2.updateString("fooField", "Foobar Value");
rs2.updateRow();

ResultSet rs3 = stm3.executeQuery("SELECT * FROM foo");
rs3.absolute(1);
con.commit();
System.out.println(rs3.first()); //true
```

Local transactions All JDBC™ compliant drivers are required to include local transaction support. Transaction management in JDBC™ API follows the SQL99 specification with *auto-commit mode*, *transaction isolation* and *savepoints* concepts. There is no way to explicitly *begin* a new transaction; a new SQL statement that requires a new transaction implicitly starts a new one, if there is no open transaction. All transaction operations are launched through the `Connection` interface, including the auto-commit policy (that is `true` by default).

Auto-commit mode — when `true` then any execution of a statement implies changes to real data because the statement is committed; when set to false an explicit commit operation must be invoked in order to make changes to the real data source.

Transaction isolation levels — defines the following three behaviors

dirty reads that occur when

- Transaction `FOO` changes a row (no commit)
- Transaction `BAR` reads the same row and sees non committed data

non-repeatable reads that occur when

- Transaction `FOO` reads a row
- Transaction `BAR` changes the same row
- Transaction `FOO` reads the same row and sees differences

phantom reads that occur when

- Transaction `FOO` reads all rows that satisfy a condition
- Transaction `BAR` adds a new row that satisfy the same condition
- Transaction `FOO` reads all rows that satisfy the same condition and sees the new row

As for `ResultSet` a set of constants has been defined to be passed to `Connection`'s `setIsolationLevel` method in order to switch from a policy to another. Table 2 summarizes those constants.

Savepoints — provides a step-by-step control on transactions by allowing different intermediate points to be saved for rollbacks, if needed. Several savepoints could be created and released through dedicated `Connection`'s methods; each savepoint is identified by a `Savepoint` object. The following sample code shows a clarifying example of use (the existence of connection and statement objects, `con` and `stm`, is assumed)

Constant	**Behavior**
`Connection.TRANSACTION_NONE`	the driver is not JDBC™ compliant since it does not support transactions
`Connection.TRANSACTION_READ_UNCOMMITTED`	dirty, non-repeatable and phantom reads are allowed
`Connection.TRANSACTION_COMMITTED`	prevents dirty reads but non-repeatable and phantom reads are allowed
`Connection.REPEATABLE_READ`	prevents dirty and non-repeatable reads but phantom reads are allowed
`Connection.TRANSACTION_SERIALIZABLE`	prevents dirty, non-repeatable and phantom reads

Table 2: A table that summarizes all transaction isolation level related constants and behaviors.

```
//Create first savepoint
Savepoint foo_svp = new Savepoint("FOO_SAVEPOINT");

//Make a(n uncommitted) change
int rows = stm.executeUpdate("INSERT INTO " +
                     "foo VALUES(‘my‘, ‘bar‘)");

//Create second savepoint
Savepoint bar_svp = new Savepoint("BAR_SAVEPOINT");

//Rollback to the second savepoint
con.rollabck(bar_svp);

//Rollabck to the first savepoint
con.rollback(foo_svp); //or con.rollback();

//Commit (the current savepoint, the first)
con.commit();
```

If a rollback procedure is invoked without a savepoint argument, the transaction is rolled back to the initial situation.

Advanced features This paragraph briefly lists some JDBC™ interesting advanced features. Each feature is not fully explained; this will be done in next steps – after a closer investigation on PoLiDBMS – when/if *advanced features* are chosen to be supported by the driver.

MetaData information — provides some meta-data information to be held in dedicated objects in order to describe all data source behaviors, capabilities and supports. With such information, a developer is able to retrieve many details about the data source of interest.

(Customizable) Type mapping — it defines type mapping from SQL types to the Java programming language types. (It allows an application to define its own type map that may differ from the default one).

Connection Pooling — it achieves performance enhancement through connection caching and reusing. Since the performance bottleneck is the connection creation, a connection object could be saved and reused when a new one is requested.

Distributed Transaction — as mentioned above it provides the support for distributed transactions through `XADataSource`, that is the `DataSource` extension that supports distributed transaction control with a two-phase commit protocol.

Retrieval of auto-generated keys — allows an application to retrieve those auto-generated keys that in some cases are generated whenever an `INSERT` statement occurs.

Multiple open `ResultSet` objects — it allows to keep an old result set instead of destroying it before returning a new subsequent one.

Batch Statements — after setting auto-commit mode to false, this feature allows to create a set of statements to be sequentially executed from the first to the last.

RowSet — it is a particular `ResultSet`'s extension that makes it JavaBeans compliant by supporting JavaBeans properties and following the JavaBeans event model. A `RowSet` is allowed to establish its own database connection to fill itself with data by normally standing *over* the driver.

2.1.2 Recommendations and Testing

Let us now illustrate how a developer should proceed when planning to implement his/her own driver.

Sun® suggests to keep in mind the *requirements for all drivers*, *additional requirements*, *permitted variants* and *security recommendations* when planning to write a new driver and also during all the development process [16]. The following guidelines apply to all JDBC™ API implementation (1.0, 2.0., 3.0)

i) Drivers must support escape syntax on SQL statements.

ii) Drivers must support (local) transactions.

iii) Drivers must support for all those features provided by the underlying data source, even if this implies a JDBC™ API extension.

iv) If a given feature is not supported by the DBMS, the driver is not required to support it.

v) If a driver does not support for one feature, the given feature must throw an `SQLException`.

vi) `DatabaseMetaData` describes the database; if something is told to be supported, it must be supported and if something is unsupported, `DatabaseMetaData` must say so.

From these guidelines, the importance of meta-data classes is enforced; `DatabaseMetaData` is better than a user-manual, it is the most trusted object of the whole driver. Sun® suggest the implementation of all main meta-data classes in order to encapsulate all DBMS characteristics in a powerful and meaningful way. As an additional requirement, all driver operations are required to be multithread-safe because some objects — like `DataSource` — are concurrently accessed since they are shared.

Because of the variation in database functionality and syntax, JDBC™ allows some variations in the driver implementations; again, this can be specified by developers through both `DatabaseMetaData` and, if certain functionality are not supported, through `SQLException` throwing. On the contrary, if the DBMS has a non-JDBC feature, this can be surely implemented with the Java extension mechanism.
Security issues are related to permission checking, that should be performed at connection time, and to security access, that should be controlled by a proper security manager; this is especially true for those drivers that uses TCP connections. Driver writers are also advised to pre-fetch data from the DBMS, to cache and reuse objects and provide "finalize" methods in order to avoid a statement to remain opened and unused.

Sun® has released two JDBC™ Test Suites (1.2.1 and 1.3.1) in order to allow developers to verify their drivers' compliance. Passing the test suite indicates that a driver is *compatible* with other products that conform to the J2EE specification, but this does not mean that the driver can be certified: only an external testing organization can do that.

2.2 JDBC™ on small DBMSes

"[...] smart cards have severe hardware limitations (very slow writes, very little RAM, constrained stable memory, no autonomy, etc.) [...]. The major problem is scaling down database techniques so they perform well under these limitations."[5]

A methodology for Very Small DataBases has been developed [6] using the same philosophy and a portable DBMS has been created, PoLiDBMS [8]. The current project is based on a similar paradigm, that is

"The major problem is scaling down JDBC™ technology so it perform well under PoLiDBMS' constrains."

and the following section will investigate this issue and discuss our point of

view.
Previous sections presented JDBC™ in general; here we wish to discuss how this technology should be scaled down to be adapted for small DBMSes; the discussion is mainly focused on the existing experience of PoLiDBMS. To conclude, necessary features for the PoLiDBMS JDBC™ layer are summarized; this topic will be completed in the requirements' analysis and specification steps (3).

We have taken a procedural and methodological approach to the *scaling down problem*; after a brief literature [5, 6, 7, 17, 3, 11] overview about *very small databases* – on both the hardware side and the software side; all most critical parts have been tracked w.r.t. a JDBC™ driver realization.
Let us now illustrate the result of our study, shown in Figure 8; identifying main steps we followed to *scale down* JDBC™ technology for PoLiDBMS. This procedure is *not* meant to be *complete* or *general purpose*; it is intended to be a *prototype* we want to test on a real case.

The first thing that should be taken into account is that a portable DBMS has not a fixed role in a hypothetical distributed system; it is an *actor* ([9]), that means it could be either a *server* or a *client*; the JDBC™ driver should enforce and natively support this particular role of the DBMS letting applications developers not bother this aspect.
The goal of a portable DBMS – that usually runs on PDAs – is to organize and manage personal data and information (*Single-application databases* or *Personal (micro) information systems* [7]), this means that no large objects or highly structured data types are used or supported by the underlying data source; this implies that the JDBC™ driver implementation has to ignore particular data types and cut-off every complex type by supporting only a minimal – but sufficient – types set. Note that this does not mean that future extensions will be denied; the support for plugging in new features should be provided in every part of the driver.
On *"Logical and Physical Design Issues for SmartCard Databases"* [7], power requirements issues are identified as critical; the JDBC™ driver – as a running software that requires certain system's resources like CPU and RAM – should be

frugal about this. The driver developer has to take care of every unknown 3rd party algorithm, he/her should investigate about the complexity of each procedure, avoiding uncontrolled cycles, preferring those data structures that are suggested by the specification of the Java programming language and – if it is applicable – use of object caching and reuse.
Driver developers should not only know about DBMS' boundaries but they have to deeply analyze the entire DBMS' internal structure; in this way, for each *high-level* request what happens at a *lower-level* is always well-known, from the parser to the core.

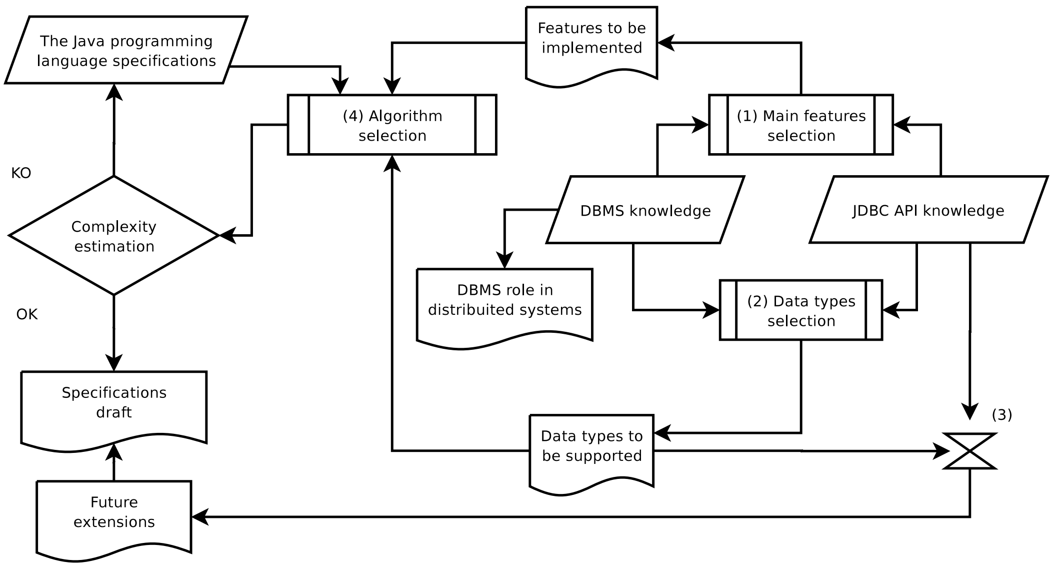

The Java programming language specifications
Features to be implemented
(4) Algorithm selection
(1) Main features selection
KO
Complexity estimation
DBMS knowledge
JDBC API knowledge
OK
DBMS role in distribuited systems
(2) Data types selection
Specifications draft
(3)
Data types to be supported
Future extensions

Figure 8: JDBC™ on small databases

2.2.1 JDBC™ for PoLiDBMS: the vision

This section illustrates the project vision. Figure 9 shows the distributed environment on which the PoLiDBMS is collocated; any of the displayed PDA has a PoLiDBMS instance on it. Beginning from this picture, we will now details our vision over the entire project, that is *"to get those arrows properly working"*.

Let us to enumerate the most significant points of the project

Mission — realize a prototype implementation of a JDBC™ driver for PoLiDBMS, that is a small DBMS.

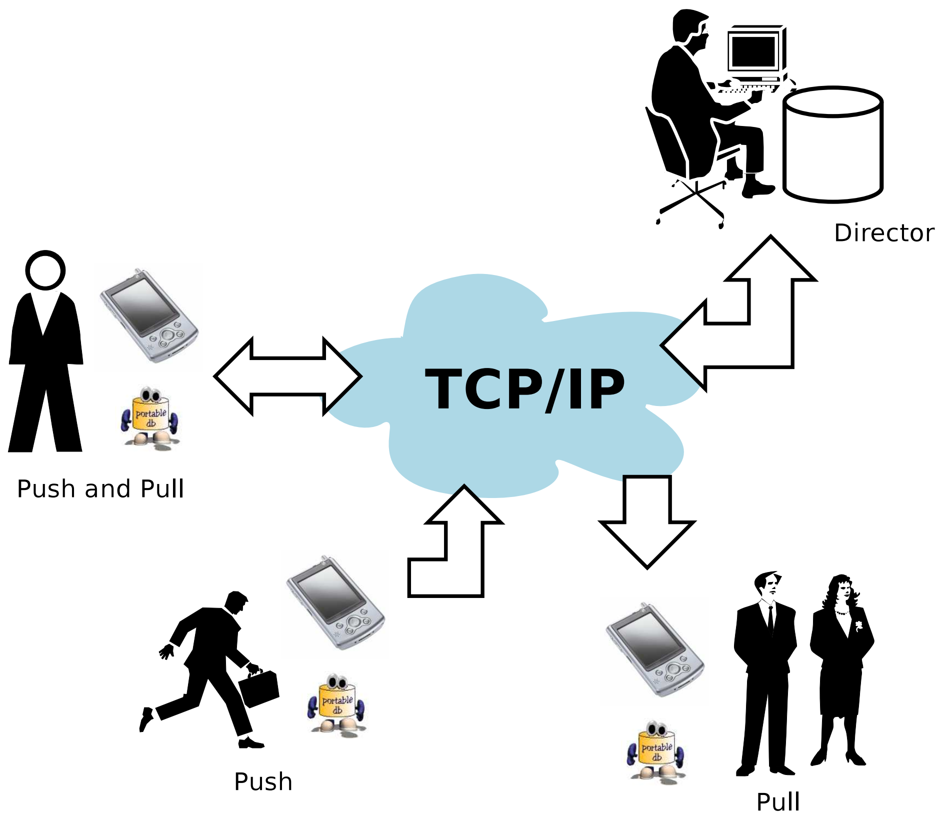


Figure 9: The hypothetical distributed scenario assumed by the project vision.

Scaling down procedure — Figure 8 will be followed, in order to test its feasibility, correctness and possibly enriching it for the future.

Project's output — the project will provide a *standard* and *transparent* interface to a customized DBMS.

Performance — the JDBC™ driver's response time will be as short as possible.

Technology — the project will *get only the best* from the JDBC™ API so performance problems will be avoided. With *'the best'*, the most thin and clean parts are intended.

Compliance — because of the particular approach, the JDBC™ driver could be a non-fully compliant one. This does not mean that the driver will not be standard, this implies that the *mission* is more relevant than the *compliance.*

Requirements — the requirement analysis will take into account every possible extension for future development.

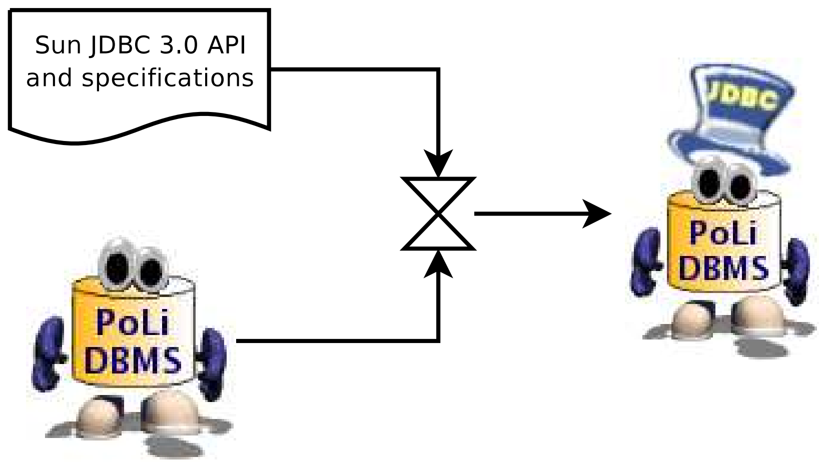


Figure 10: The mission

Implementation — the specification and the implementation steps will guarantee software modularity to make future development as easy as possible.

Approach — an hybrid approach will be followed. The driver will have the *non-obsolescence* and the *advantages* of the `DataSource` interfacing approach, together with the *simplicity* of the `DriverManager` method. Meta-data classes will be partially[2] implemented, in order to use them now and in the future as a development status tracker.

Testing — the existing JDBC™ Testing Suite may not work since the resulting driver could be a non-fully compliant one. Custom test cases will be created.

PoLiDBMS with a JDBC™ layer (use cases) This paragraph shows how PoLiDBMS with a JDBC™ will appear in terms of *how it will be used.* This topic has been partially treated while presenting the JDBC™ 3.0 API, with the related code sample; let us to depict a common scenario that involves a PoLiDBMS instance that is used both locally – by the PDA owner – and remotely – by the system administrator (Figure 11–12).

Figure 11 shows that different users can benefit from a JDBC™ layer with several use cases that can involve multiple actors to interact on the same data source in a transactional environment. In Figure 12, the transactional fashion is enforced; the DBMS is assumed to be transaction safe and to support multiuser

[2]There are several methods that are thought for big DBMSes, due to this they cannot cope with our little DBMS

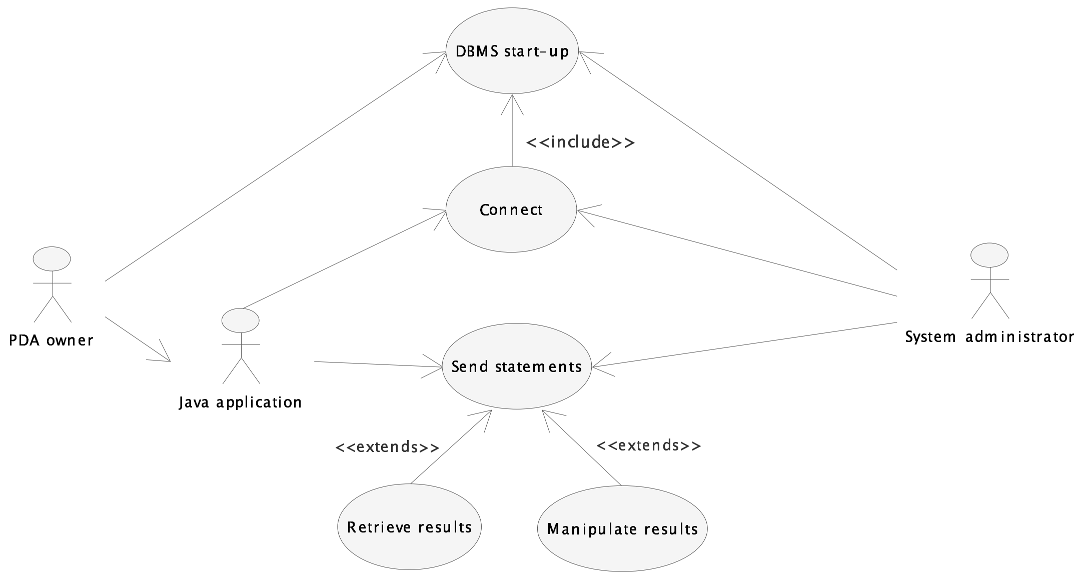


Figure 11: A use-case diagram showing how PoLiDBMS can be accessed by different actors using a JDBC™ layer.

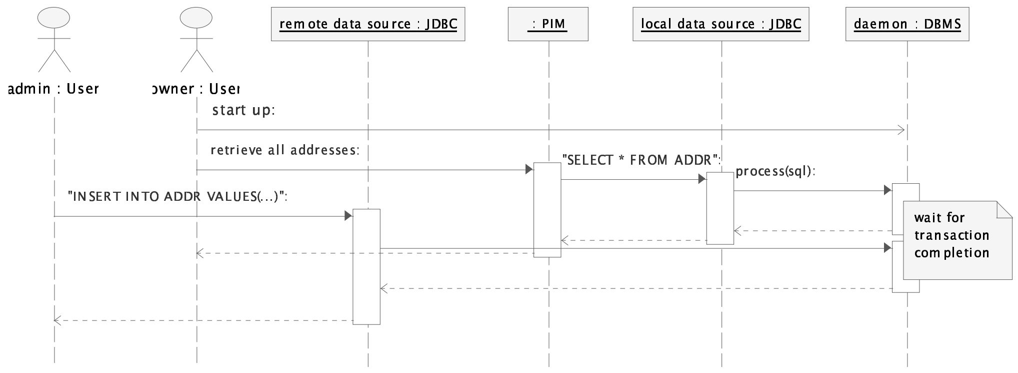


Figure 12: A sequence diagram showing how PoLiDBMS can be accessed both locally and remotely using a JDBC™ layer.

access.

Chapter III

Design requirements & specifications

The first sections of this chapter presents a general summary of the development background, that is the PoLiDBMS status, its features and its primary parts where the requirement analysis has focused on. In the following sections the requirements' analysis is presented with some use case scenarios in order to take into account every possible application interaction with the underlying DBMS. The chapter discusses the design specification, in natural language along with some significant UML diagrams.

3.1 Background analysis

In order to identify the design requirements, after having explored PoLiDBMS through console interface, the following section shows the results of this study.

3.1.1 PoLiDBMS high level structure and main features

Figure 13 shows an intuitive representation of the structure of PoLiDBMS. The first peculiar thing is that the DBMS has a highly modular architecture; all functions and roles are well separated from each others, this is especially true for the *query processor* package, that is the *real* DBMS. This modular structure allowed all boundary modules to be easily *pluggable*; thanks to the well specified *query processor*, the upper and the lower layers have been independently thought and developed and this helps new features – like JDBC™ – to be welcome.

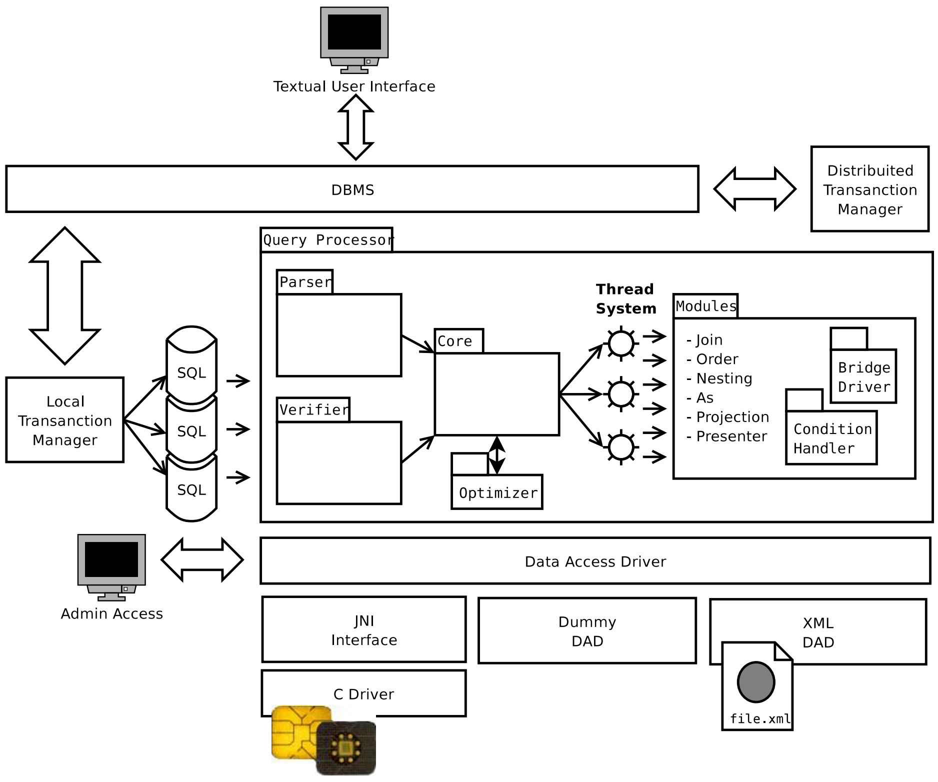


Figure 13: A high level representation of the DBMS' structure

Since the project's main goal is to attach an upper layer and mask the whole DBMS structure, the most interesting parts constraining our project are located on the top of the architecture; more precisely on

Parser — as a common SQL parser, this module is in charge the processing of an SQL and it does this by a three-step workflow; *lexical analysis*, *syntactical analysis* and *coherence control*. Since JDBC™ purposes are not focused on SQL validation or query parsing, the most relevant aspect of the parser is the supported SQL and the BNF (Backus Naur Form); this is because JDBC™ has only fully-featured interfaces (like `Statement`) and *what to be implemented* must be known [2].

DBMS — at this time, 'DBMS' is also the classname of the unique interface to the *local transaction manager*. Thanks to its methods (like `begin`, `commit`, `process`, etc.) this `DBMS` object can be used to:

- send a login request
- send queries to the query processor

- begin, commit and rollback a transaction
- get back the results

This may become a good *'plugging point'*, that is where the driver could be attached [8].

Local Transaction Manager — it communicates both with the query processor and with the DBMS layer; it manages any single transaction or query in order to preserve ACID (Atomicity, Consistency, Isolation, Durability) properties. Due to this, the local transaction manager has some rules that must be strictly followed to make it working properly [1]:

- any transaction, even if it is a query, must be preceded by a *begin* statement and followed by a *commit* statement; no overlapping is allowed;
- it does not accepts queries as strings, any SQL statement (DML or DDL) must be *encapsulate* into a so called; `ServicePack` object (it can be of cardinality 1 or more);
- the above two procedures can only be executed by passing through the `DBMS` interface, as already discussed;
- it does not supports intermediate/multiple savepoints; if a rollback procedure is invoked, the transaction is rolled back at the beginning point (it acts as a *'backup'*).

3.1.2 PoLiDBMS critical parts and development status

We classify a part (module) to be *critical* (w.r.t our interest) when

It is under development — this means that it should be tested or that it cannot be a trusted segment of a procedure. Attention must be paid to its error signals or exception conditions;

It is a boundary entity — this means that its code should be trusted; it is treated as an interface and because of this it will be used to *'talk'* with the DBMS;

It does not exist (but is needed) — this implies that a *make or buy* analysis must be done in order to decide if *we should try to make it* or *it would be better if we ask for it* (and make a *dummy* module).

We also found that all data is organized into an unique logical database; no multiple database are supported and no catalogs have been found. This is consistent with the general scenario of a personal, portable, small device/database.

3.1.2.1 PoLiDBMS interesting parts

As already mentioned PoLiDBMS is under development; some parts like the core, the parser, the data drivers and the transaction system have passed some testing sessions and they can be trusted. Furthermore, there is an interesting boundary entity (`DBMS`) already identified as a good *attach point candidate*; the importance of this topic will be highlighted again in the following requirement analysis where the necessity for a daemon will be discussed.

3.2 Requirement analysis and specification

A good requirements analysis should start from a close investigation of the *'business logic'* and of *'what stakeholders want'* (*Conallen* [12]); our *'business logic'* is of course PoLiDBMS, its functionalities and its features; we are *'stakeholders'* along with every hypothetical application developer. PoLiDBMS has been analyzed (3.1) and the developers' point of view has been presented in Chapter 2. The resulting requirements are here in the following discussed.
Figure 14 summarizes the main activities (see Figure 6 for all project activities).

3.2.1 Functional and Architectural requirements

Here we illustrate a structured representation of the functional and architectural requirements. They do not follow a particular order; when there is an indentation level this means that the indented requirement is implied by the above one.

3.2.1.1 High level

We start by viewing things from a high point of view; here we deal with major issues and with all aspects that are visible from the outside the driver. Issues like *language*, *platform*, *environment* and more are here dealt with.

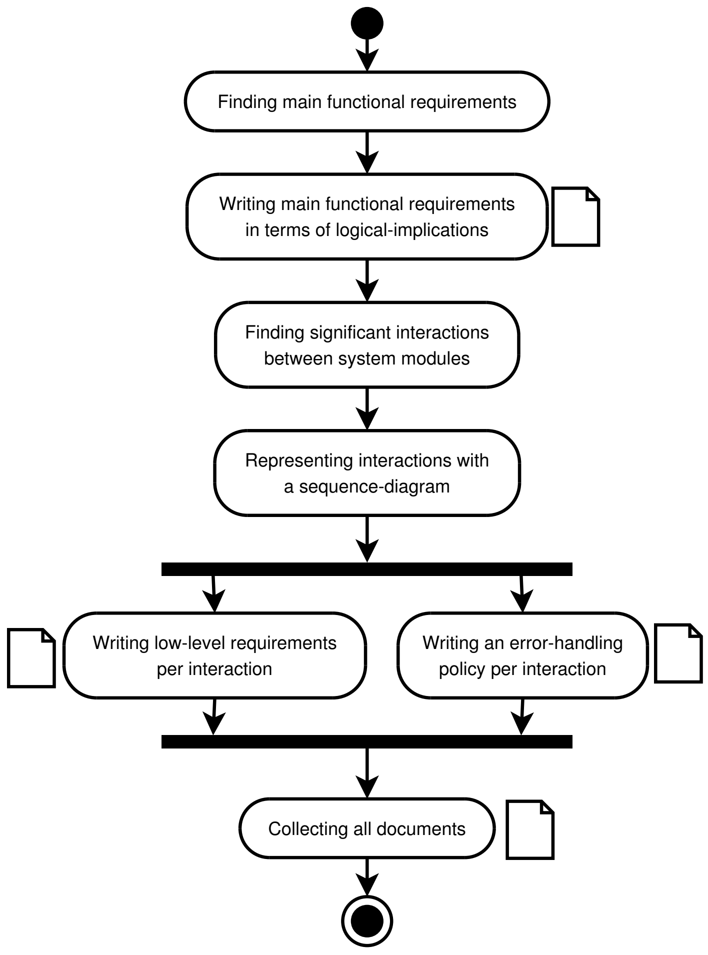

Finding main functional requirements
Writing main functional requirements in terms of logical-implications
Finding significant interactions between system modules
Representing interactions with a sequence-diagram
Writing low-level requirements per interaction
Writing an error-handling policy per interaction
Collecting all documents

Figure 14: Requirements' analysis main activities

1. the DBMS is written in Java ⇒
   (a) no need foreign language interfacing;
   (b) platform independence.
2. the DBMS is for portable devices ⇒
   (a) attention to *space* issues;
   (b) attention to *CPU resources* issues;
   (c) attention to *execution time* issues.
3. JDBC™ is a standard, big and well specified API ⇒
   (a) possibility of implementing almost anything we need;
   (b) select what features to implement
   (c) cut out a small subset of the interface

4. the DBMS only has a *console access* ⇒
   (a) necessity of a *plugging point*
   (b) necessity of a daemon to make PoLiDBMS fully reachable
   (c) PoLiDBMS should have at least one access point
      i. particular attention to security issue
      ii. the daemon will be the unique *plugging point*
5. sometimes the data is remotely accessed and (4b) ⇒
   (a) the daemon must be a network daemon (little server)
   (b) the network is a TCP/IP and Java has the Remote Method Invocation (RMI) framework
      i. the daemon will be reached by a logical URI
      ii. use of RMI to start up a daemon and bind it to a logical (unique) network name
6. the DBMS preserves the ACID properties for each operation ⇒
   (a) the driver must grant ACID properties for single operations and transactions
7. the DBMS does not support multiple logical databases and catalogs ⇒
   (a) the driver has taken into account that all the data are organized into a unique database with no catalogs

3.2.1.2 Low level

By knowing how PoLiDBMS works, we now experiment some interfacing scenarios[1] from which we build the main interface requirements. Since there is no ordinary *system users*, each interaction is considered between two or more nodes of the whole system. The most important interactions we chose are; *start-up the DBMS*, *login*, *send a statement*, *get back the results*, *begin, commit and rollback procedures.* The following scenarios are explained assuming *no error conditions*, thus *errors* are considered separately. No particular order has been followed.

[1]as Liskov [4] suggests, scenarios for typical interactions are useful for functional requirements

Daemon start-up Assuming that we can reach the *DBMS* through a single class (`DBMS`) and we can interface to the *Transaction Manager* only by sending it some messages, PoLiDBMS daemon should behave as follows (Figure 15 shows the related sequence diagram).

1. instantiate a new `DBMS` class
2. tell the transaction manager to begin a recover procedure to restore all necessary data
3. tell the RMI registry to bind the `DBMS` instance to a logical name
4. wait to the recovery procedure to be completed

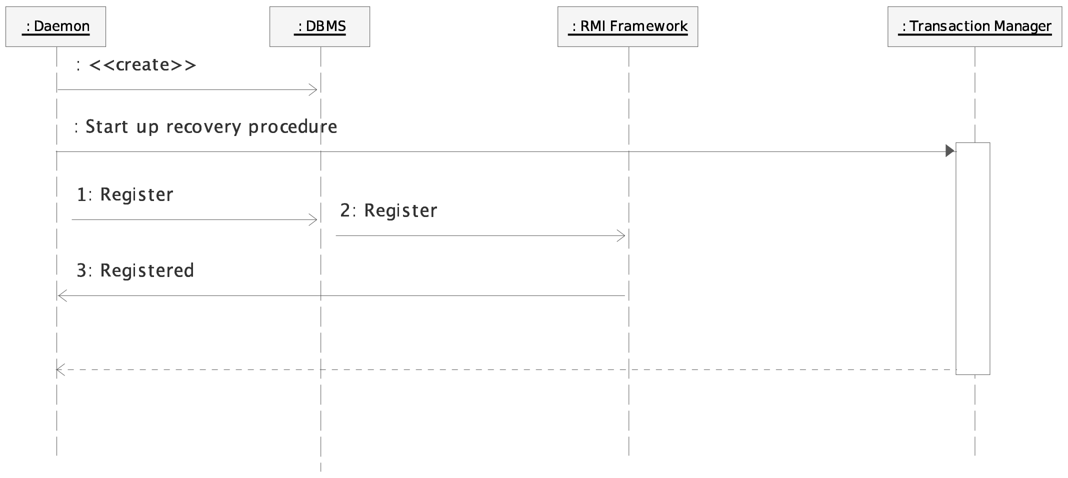


Figure 15: Sequence diagram for starting up the DBMS daemon

Daemon start-up (error handling) Any error is here considered as *fatal*; if something fails, the DBMS cannot start and this implies that the daemon cannot serve any request because of this absence. These are the most important errors

1. the transaction manager fails on recovery procedure
2. RMI fails when adding/binding the `DBMS` object to the registry

in the above cases, the corresponding exception signals should be caught by the daemon which must *halt* and then *report* any problem to the standard output.

Login procedure This is assumed to be done by a generic node (in our specific implementation this operation may be done by either a `DataSource` or `Connection` implementation, but it does not matter at this point). The node may do what follows, as also explained by Figure 16 with a sequence diagram.

1. a `DBMS` reference is registered to the daemon, by specifying it a known URL;
2. with the `DBMS` reference, the `login` method can be called;
3. the `DBMS` might forward the login request to a specific handler and then reply; to the caller appropriately (with a `boolean` value).

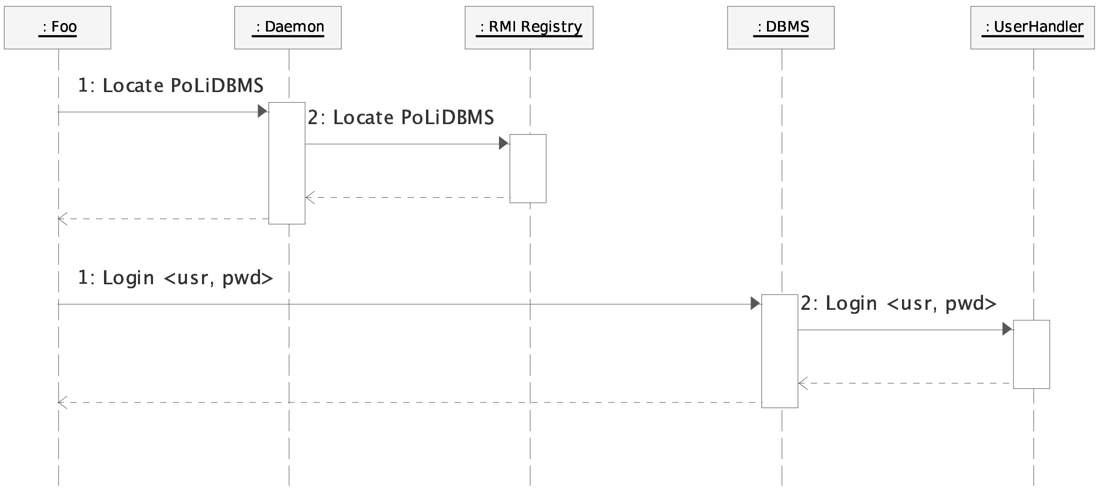


Figure 16: Sequence diagram showing how the user authentication interaction (intended to be performed by a generic node)

Login procedure (error handling) There is a *fatal* error and a login error; the first occurs when something fails with RMI or the URL is mismatched. The latter could happen when the `<username, password>` couple is not a valid one. In the first case the caller should *halt* because it does not have any valid `DBMS` reference; in the second case a simple exception signal can notify the caller of that problem and the corresponding code can be easily corrected.

Send a statement and get back the results Following the JDBC™ specifications, some steps must be followed.

1. login request (treated in the last paragraph);

2. instance a new `DataSource` implementation;
3. use the `DataSource` to get a `Connection` object;
4. use `Connection` to create a new `Statement` object;
5. send an SQL statement through the `Statement` object.

Since these steps are already well specified, we analyze what any single implementation should do to correctly serve such a request. We assume what follows

1. the login procedure has been properly completed with success;
2. the `DataSource` object has been already instantiated;
3. the `DataSource` object has a `DBMS` reference;
4. the `Connection` object knows how to find its related `DataSource`.

Starting from the JDBC™ specifications and from the above hypotheses, we start from the `Connection`'s point of view. It has to

1. instantiate a `Statement` object that is related to it;
2. from the `Statement` object, an *execution* method could be called by passing it the SQL statement;
3. the `Statement` validates and forwards the call to its own `Connection`;
4. the `Connection` redirects the call to its related `DataSource`;
5. the `DataSource` calls the `DBMS` that processes the statement and returns back the results (optionally, the `DataSource` object could validate the `Connection` that called it);
6. the above sequence is followed in the opposite order to return the results to the caller (`Statement`).

All the above procedures are explained in Figure 17. While analyzing PoLiDBMS, we determined that *any query is a mono-operation transaction*; because of this, any single-statement must be preceded by a *begin* and followed by a *commit*.

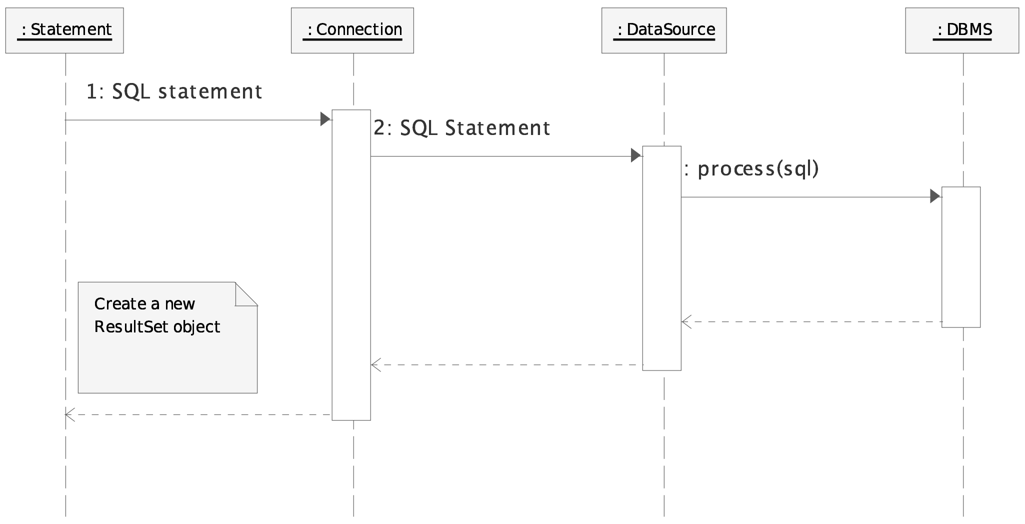


Figure 17: A sequence diagram showing how a `Statement` object accepts and forwards an SQL statement

Send a statement and get back the results (error handling) Common errors within these interactions are especially related to the *transaction manager*, which could fail, and to the *submitted query*, which may contain syntax errors. The first case should be treated as a *fatal error* because if the transaction manager is not able to handle a particular transaction/query it is true that some data can become inconsistent or damaged; the second kind of errors is not really a software problem and it can be handle by informing the user and ignoring the transaction/query.

Begin, commit, rollback procedures These three interactions are very similar to the last **Send a statement** interaction. The most significant difference is the scope; a `Statement` object should *not* have those methods, only `Connection` can have them. The reason is the following one; a connection object may have more than one active `Statement` and each of these does not know about the others so they cannot safely invoke a *commit* or a *rollback*. A typical scenario explains how some statements can be sent in a transactional environment, all followed by a transaction commit.

1. A transaction *begin* command is sent;
2. Some SQL statements are sent through the `Statement` object;
3. If there are some results, they are collected;

4. A transaction *commit* command is sent.

Figure 18 summarizes the above steps.

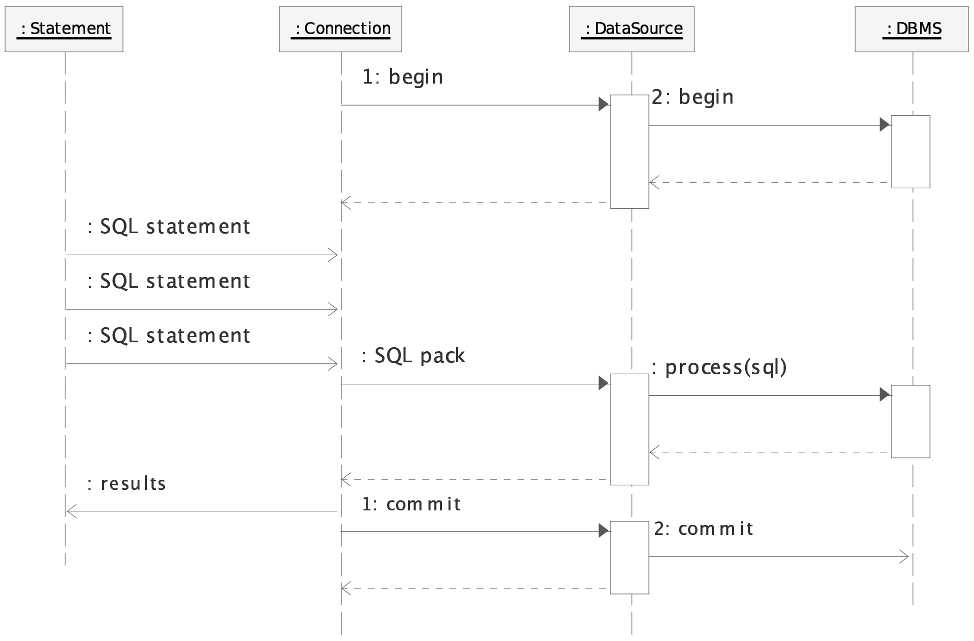


Figure 18: A sequence diagram showing a transaction example.

Begin, commit, rollback procedures (error handling) This case is included into the previously considered one (see 3.2.1.2); if there is a problem with the *transaction manager*, this should be safely treated as a *fatal error* or some data could be damaged/lost. A safe way should be checking the status of the transaction manager just before beginning the whole transaction (or query), if something is wrong with it, the entire procedure should terminate with an exception signal.

3.2.2 Non-Functional requirements

In addition to the functional requirements, a list of other desired requirements is here presented.

Fast — portable applications are thought to be fast and *click 'n find*; the end user does not want to wait too much for his/her data. This means that the driver does not to be the *bottleneck*; the DBMS processing capacity must be preserved;

Usable — a JDBC™ driver is also commonly and widely used by non-expert programmers that uses it for simple data-retrieval applications (like calendars, PIM, organizers, etc.). This is highly guaranteed by JDBC™ API;

Work In Progress — since the DBMS is a *Work In Progress*, the driver is a *Work In Progress* too; it should be modular, open, simple and easily testable by other developers for future enhancement or other features implementation.

They are not strictly related to the implementation work but some of they (**Fast**) have an impact on the chosen algorithms and data structures.

3.2.3 Design specifications

By starting from the constrains identified through the requirements' analysis, the design specification of each module has been carried out. It is important to remember that a JDBC™ driver is an implementation of the JDBC™ API (or its subset); given this premise, the whole specifications can be treated and presented w.r.t. Sun®'s interface specifications.

In Figure 19 the workflow of all activities for specifications of the whole driver and its related parts is summarized (an activity with the *paper icon* implies a written output). The main activities are Choosing what to implement, Finding unimplemented parts to beimplemented and their related formalization steps.

JDBC™ interface subset We chose the `DataSource` approach; first because it is recommended by Sun® for new flexible drivers, second because it can be easily extended with the distributed transaction support, third because the `DataSource` is a clean representation of the DBMS as a whole and finally because all the backward compatibility is preserved for each `DataSource` related object.
Starting from the above assumptions, the following list enumerates the interface subset components:

`javax.sql.DataSource`

`java.sql.Connection`

`java.sql.Statement`

`java.sql.ResultSet`

`java.sql.DatabaseMetaData`

`java.sql.ResultSetMetaData`

`java.sql.SQLException`

`java.sql.Date` (an already implemented wrapper)

`java.sql.Time` (an already implemented wrapper)

The role of each object has already been explained (see 2.1.1) so now we concentrate on designing a *"what has to be done and how should it be done"* way. Figure 20 shows the associations between the main objects.

Note (namespace) Because of self explaining reasons, we chose the prefix `PoLi-` to denote the implementation of the interface that follows that prefix. As an example; `PoLiDataSource` *implements* `DataSource`.

The implementation specifications for JDBC™ related classes follows (Note: an existing specification for a PoLiDBMS daemon is assumed).

`PoLiDataSource` ***implements*** `DataSource`

Where A `DataSource` implementation is located *client-side*; it is typically positioned between a (Java™) application and the real DBMS that is often a remote node (*server-side*).

What It represents the whole DBMS; it has to create/release connections; it should be able to maintain a pool of (prepared) connections to reduce connection creation time; it has to provide methods to get connections with and without `<username, password>`.

How It connects to the DBMS though a daemon over a TCP/IP; it uses an RMI registry to locate the daemon with its URI; its main methods must be synchronized in order to be able to serve multiple requests; it has to provide some methods to wrap query/transaction processing from a `Connection` to the DBMS (Figure 21); it has to keep track of all created connections.

Problems Connection failure handling; report all RMI related problems; wrapping exceptions propagation; if a method is not implemented yet an appropriate exception type must be thrown; avoid foreign calls of the wrapper (a good way is to leave those wrappers `protected`).

Classname `it.polimi.elet.dbms.jdbc.PoLiDataSource`

Related `it.polimi.elet.dbms.PoLiDaemon`,
`it.polimi.elet.dbms.DBMS`

`PoLiConnection` ***implements*** `Connection`

Where It is located in the same place as a `DataSource` implementation; it is logically located between a (Java™) application and a `DataSource` implementation.

What It logically represents an established connection (session) with the DBMS; it should be distinguishable from other connections, a simple unique `DataSource`-generated key is enough; it has an unique related `DataSource`.

How It must provide methods to wrap query/transaction processing from a `Statement` to the related `DataSource` (Figure 21); it could be closed (a good way to do this is to clear any reference to the related data source and leave object destruction to the garbage collector); error/warning logging is not supported;

Problems If a method is not implemented yet an appropriate exception type must be thrown; avoid foreign calls of the wrapper (a good but not fully safe way is to leave those wrappers `protected`).

Classname `it.polimi.elet.dbms.jdbc.PoLiConnection`

Related `it.polimi.elet.dbms.jdbc.PoLiDataSource`

`PoLiStatement` ***implements*** `Statement`

Where It is located in the same place as a `Connection` implementation; it is logically positioned *'into'* a (Java™) application.

What It represents a generic SQL statement (DDL or DML); formally it is an object that provides a way to send statements to the DBMS and either get back the results or retrieves meta-information.

How It it generated by a (valid) `Connection` object; it must wrap (Figure 21) a statement processing through appropriate methods up to the related `Connection` object; it has to provide a timeout between send–receive instants; it should be closeable.

Problems If the sent–receive timeout expires the whole query/transaction must be completed and safely rolled back; if no problems occur, a statement must be always completed and it cannot be canceled.

Classname `it.polimi.elet.dbms.jdbc.PoLiStatement`

Related `it.polimi.elet.dbms.jdbc.PoLiConnection`

`PoLiResultSet` ***implements*** `ResultSet`

Where It can be generated by a `Statement`'s methods so it stands in the same 'place'.

What It represents a set of results, meant to be a set of records, organized into a cursor-pointed tabular structure; it must provides ways to access and manipulate results, row-by-row; it must provide methods to *get* all supported data types and methods to *update* at least one data type.

How It is mainly based upon the existing result set implementation; the representation of `PoLiResultSet` object is an instance of `it.polimi.elet.dbms.core.ResultSet` through the `it.polimi.elet.dbms.core.ResultSetInterface` interface; it should be closeable, scrollable and updatable.

Problems The existing result set implementation does not support particular cursor positions; an auxiliary way to support this is needed; this support must have a dedicated test in order to check the cursor position to be correct; manage all different type of result sets, about concurrency, scrollability and holdability; signal unsupported types.

Classname `it.polimi.elet.dbms.jdbc.PoLiResultSet`

Related `it.polimi.elet.dbms.core.ResultSet`, `it.polimi.elet.dbms.core.ResultSetInterface`, `it.polimi.elet.dbms.jdbc.PoLiStatement`

`PoLiDatabaseMetaData` ***implements*** `DatabaseMetaData`

Where It is generated by a `DataSource`'s method, so it is in the same 'place'.

What It has to provide detailed information about the underlying (real) data source; since it is a description class it only provides methods for information retrieval.

How Appropriate class attributes and relative getters are enough.

Problems None; this class has does nothing.

Classname `it.polimi.elet.dbms.jdbc.PoLiDatabaseMetaData`

Related `it.polimi.elet.dbms.jdbc.PoLiDataSource`, `it.polimi.elet.dbms.jdbc.PoLiResultSetMetaData`

`PoLiResultSetMetaData` ***implements*** `ResultSetMetaData`

Where It is (auto-)generated whenever a new `PoLiResultSet` is instantiated, so it is in the same place.

What It has to provide detailed information about its associated result set; since it is a description class it only provides methods for information retrieval.

How Its internal representation is the set of information that a result set can export; maintaining a reference to the associated result set and retrieving the information on demand is enough.

Problems None; this class does nothing.

Classname `it.polimi.elet.dbms.jdbc.PoLiResultSetMetaData`

Related `it.polimi.elet.dbms.jdbc.PoLiResultSet`,
`it.polimi.elet.dbms.core.ResultSet`,
`it.polimi.elet.dbms.jdbc.PoLiResultSetMetaData`

In the above specifications we assumed an existing specification for a PoLiDBMS daemon, as the following formal specification defines.

`PoLiDaemon`

Where It resides on *server-side* as a DBMS boundary interface; it is started by the system administrator.

What It is an RMI-based server; it listens on a TCP port; it is in charge to make the DBMS remotely available; default server `<name, port>` couple has to be customizable by passing parameters to the `main` method; it has to prepare the DBMS before making it available.

How It uses the RMI registry to bind the DBMS to a logical URI; the `main` method has to parse the received parameters in order to set the `<name, port>` couple; it has to invoke some status-recovery procedures on the local transaction manager; make a wrapper to the local transaction manager's recovery procedure, since this cannot be directly invoked from an external module.

Problems Status reporting on the standard output for the start-up procedure; status reporting for recovery procedures is already done by the local transaction manager; the `DBMS` is not a `RemoteObject`.

Classname `it.polimi.elet.dbms.PoLiDaemon`

Related `it.polimi.elet.dbms.core.Status`,
`it.polimi.elet.dbms.DBMS`

`SQLException` and its subclasses

Sun®'s JDBC™ specifications do not differentiate between types of errors and suggest the use of `SQLException` to signal an error. Because of this reason we decide to specify an exception hierarchy to better explain the *error cause* in case of an exception signal. An appropriate package – under the `it.polimi.elet.dbms.jdbc` – has to contain different types of exceptions to explain if

- a code fragment or a method is not implemented yet;
- a submitted parameter is not into a required range of values;
- a submitted parameter is of a wrong (sub)type;
- `null` pointer is submitted;
- a `Connection` or `DataSource` or whatever is not valid.

Formal specifications In this paragraph a formalization of all the above specification is provided by means of some UML class-diagram. These diagrams have not to be taken as the implementation since they could/must be uncompleted, due to readability reasons. In Figure 22 the packages tree is shown and in figures 23 to 27 each class is detailed.

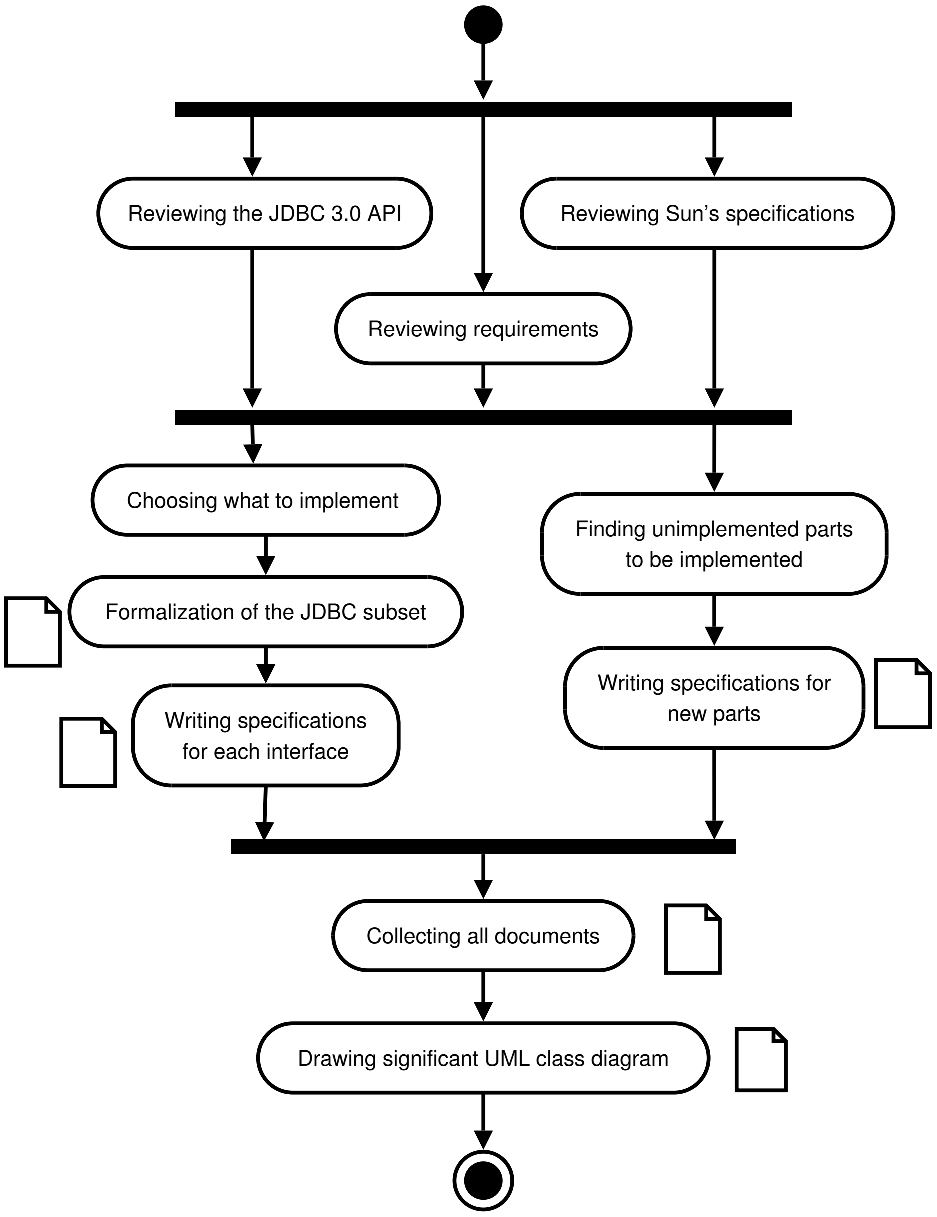

Reviewing the JDBC 3.0 API
Reviewing Sun's specifications
Reviewing requirements
Choosing what to implement
Finding unimplemented parts to be implemented
Formalization of the JDBC subset
Writing specifications for each interface
Writing specifications for new parts
Collecting all documents
Drawing significant UML class diagram

Figure 19: An activity diagram that summarizes the main activities involved in this step

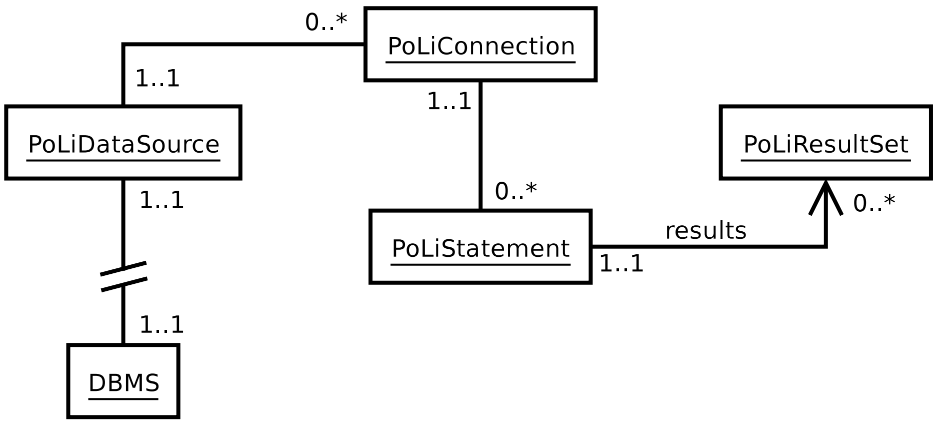


Figure 20: Associations between main objects

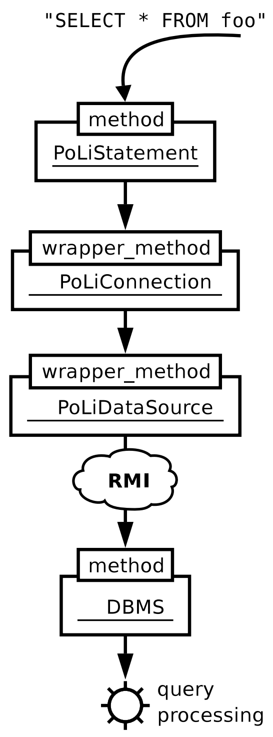


Figure 21: An intuitive diagram to show how a query process is being wrapped

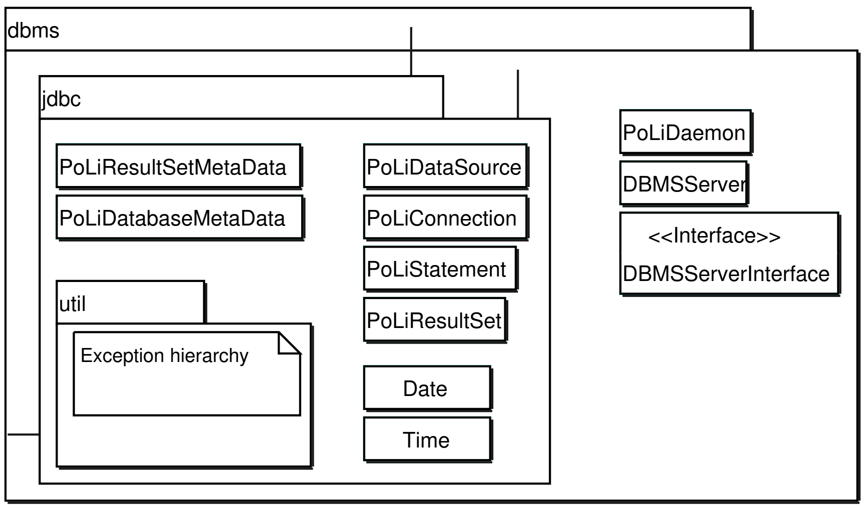


Figure 22: A class-diagram showing all packages involved in this specification. No details are shown for each class; this diagram purpose is to provide an overview of the packages tree.

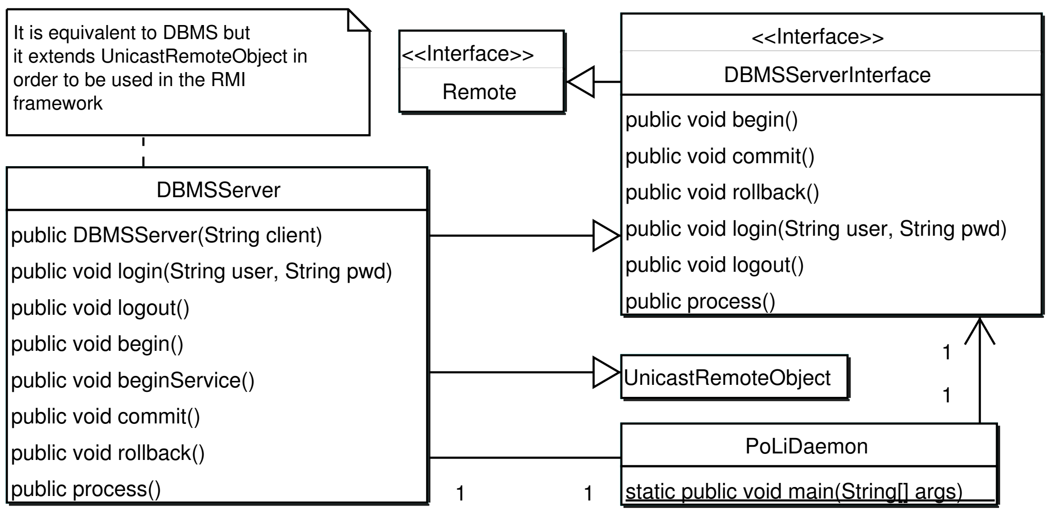


Figure 23: A class-diagram showing associations and implementation specifications for the daemon. The use of RMI is exploited by the realization of the main RMI-related class/interfaces. The `DBMSServer(Interface)` will probably substitute the `DBMS`; it is equivalent but it can be used as a remote object. The daemon has only a `main()` method that is responsible for the start-up procedure.

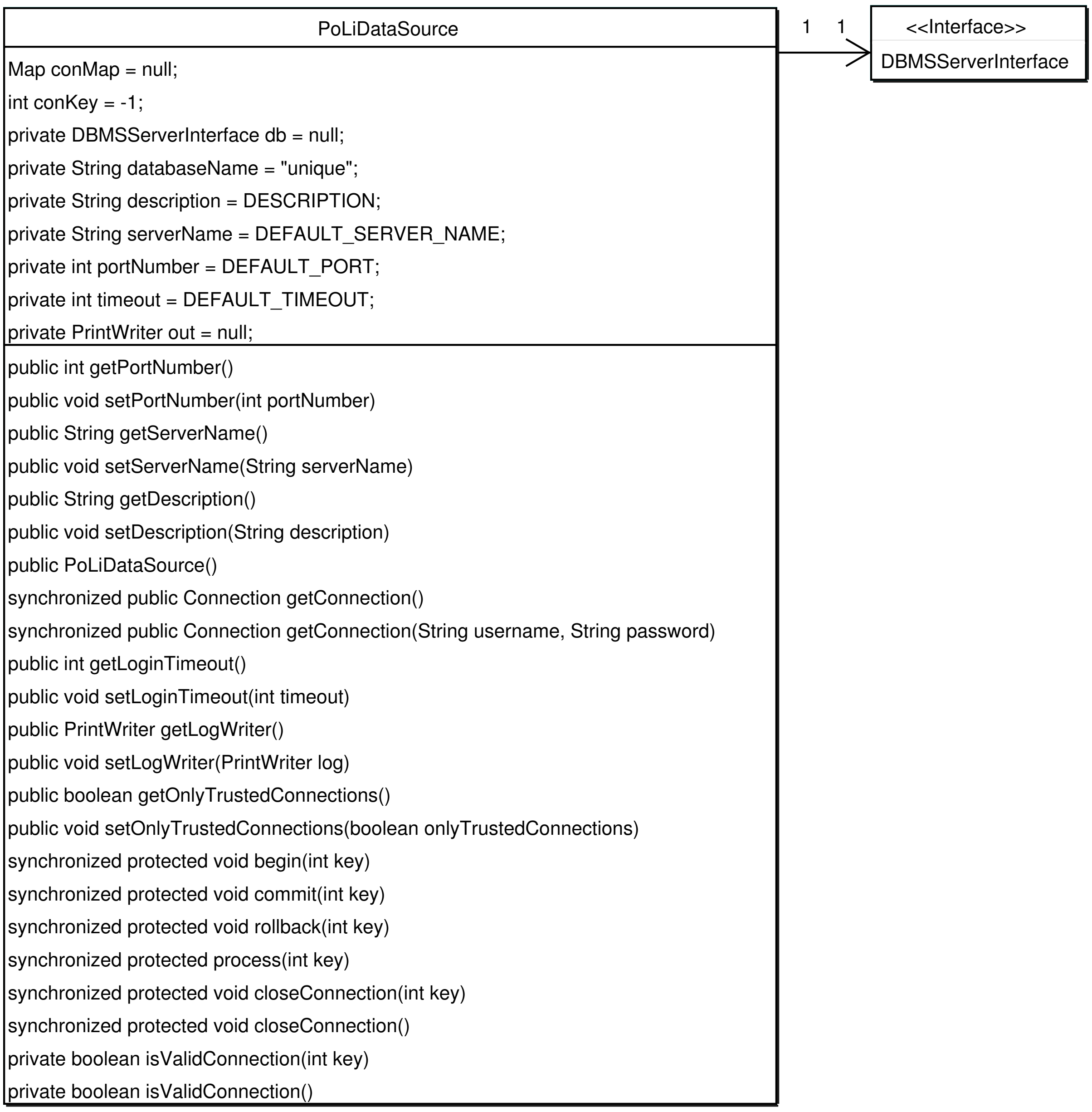

PoLiDataSource
Map conMap = null;
int conKey = -1;
private DBMSServerInterface db = null;
private String databaseName = "unique";
private String description = DESCRIPTION;
private String serverName = DEFAULT_SERVER_NAME;
private int portNumber = DEFAULT_PORT;
private int timeout = DEFAULT_TIMEOUT;
private PrintWriter out = null;
public int getPortNumber()
public void setPortNumber(int portNumber)
public String getServerName()
public void setServerName(String serverName)
public String getDescription()
public void setDescription(String description)
public PoLiDataSource()
synchronized public Connection getConnection()
synchronized public Connection getConnection(String username, String password)
public int getLoginTimeout()
public void setLoginTimeout(int timeout)
public PrintWriter getLogWriter()
public void setLogWriter(PrintWriter log)
public boolean getOnlyTrustedConnections()
public void setOnlyTrustedConnections(boolean onlyTrustedConnections)
synchronized protected void begin(int key)
synchronized protected void commit(int key)
synchronized protected void rollback(int key)
synchronized protected process(int key)
synchronized protected void closeConnection(int key)
synchronized protected void closeConnection()
private boolean isValidConnection(int key)
private boolean isValidConnection()
1
1
<<Interface>>
DBMSServerInterface

Figure 24: The class-diagram for `PoLiDataSource` with significant implementation specifications. Note that the use of the `synchronization` clause on those methods that may be called concurrently; some constant values are here omitted and minor details are moved to implementation steps; some methods are `protected` for isolation reasons.

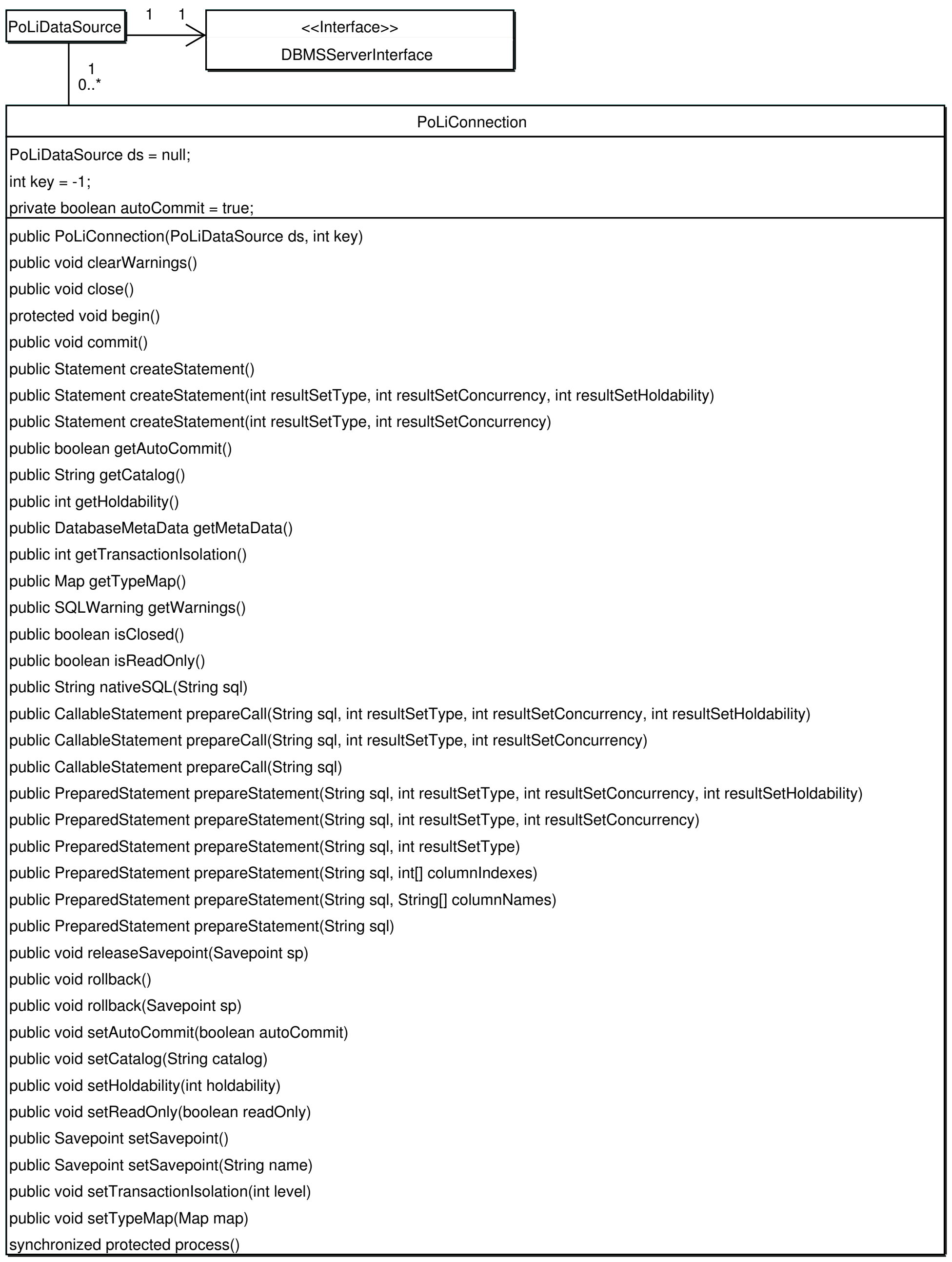

PoLiDataSource
1
1
<<Interface>>
DBMSServerInterface
1
0..*
PoLiConnection
PoLiDataSource ds = null;
int key = -1;
private boolean autoCommit = true;
public PoLiConnection(PoLiDataSource ds, int key)
public void clearWarnings()
public void close()
protected void begin()
public void commit()
public Statement createStatement()
public Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)
public Statement createStatement(int resultSetType, int resultSetConcurrency)
public boolean getAutoCommit()
public String getCatalog()
public int getHoldability()
public DatabaseMetaData getMetaData()
public int getTransactionIsolation()
public Map getTypeMap()
public SQLWarning getWarnings()
public boolean isClosed()
public boolean isReadOnly()
public String nativeSQL(String sql)
public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)
public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)
public CallableStatement prepareCall(String sql)
public PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)
public PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)
public PreparedStatement prepareStatement(String sql, int resultSetType)
public PreparedStatement prepareStatement(String sql, int[] columnIndexes)
public PreparedStatement prepareStatement(String sql, String[] columnNames)
public PreparedStatement prepareStatement(String sql)
public void releaseSavepoint(Savepoint sp)
public void rollback()
public void rollback(Savepoint sp)
public void setAutoCommit(boolean autoCommit)
public void setCatalog(String catalog)
public void setHoldability(int holdability)
public void setReadOnly(boolean readOnly)
public Savepoint setSavepoint()
public Savepoint setSavepoint(String name)
public void setTransactionIsolation(int level)
public void setTypeMap(Map map)
synchronized protected process()

Figure 25: The class-diagram for `PoLiConnection` specification. Note that the query processing wrapper method, `process()`, that is `protected` and `synchronized`; all other methods are fully well specified by the JDBC™ API.

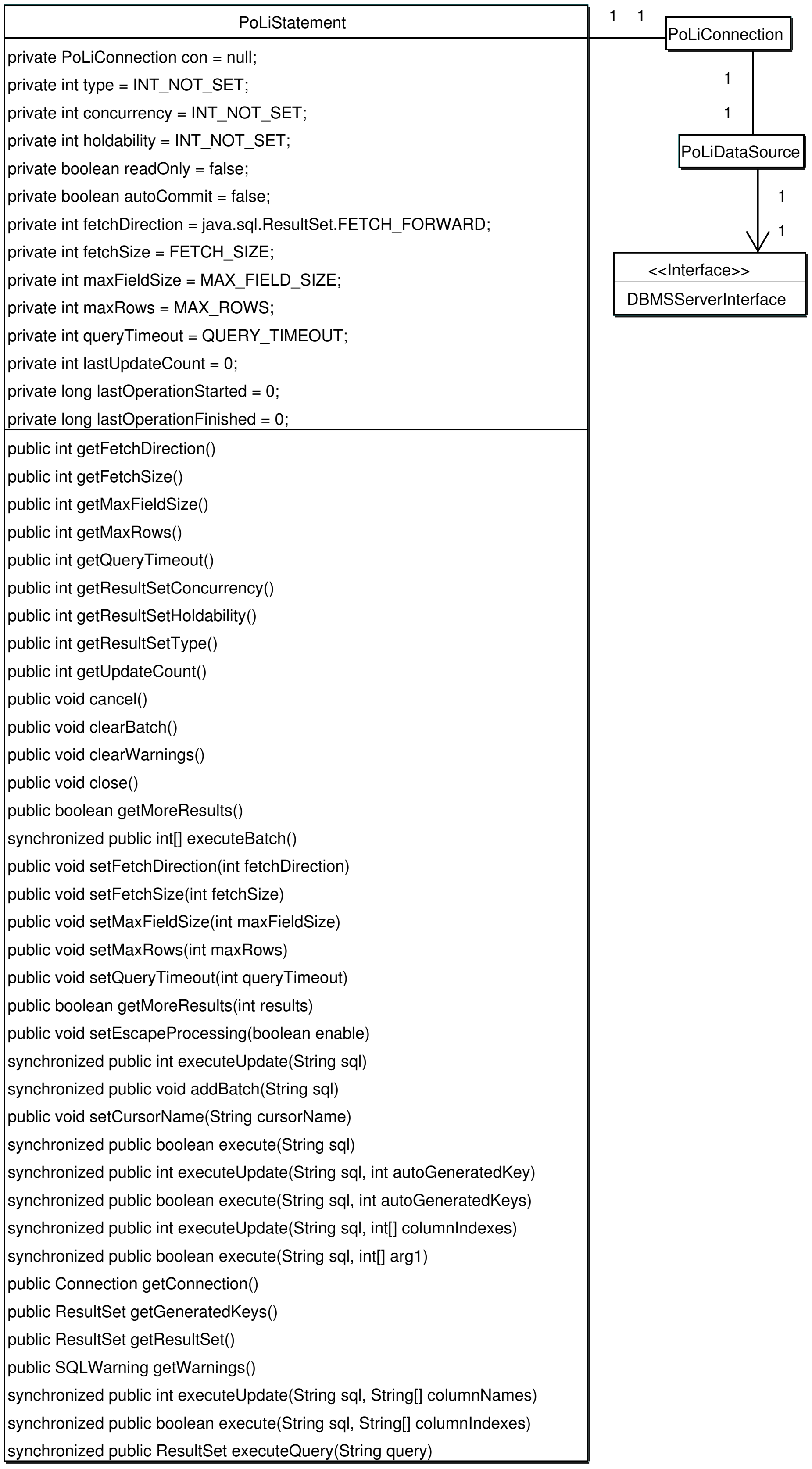
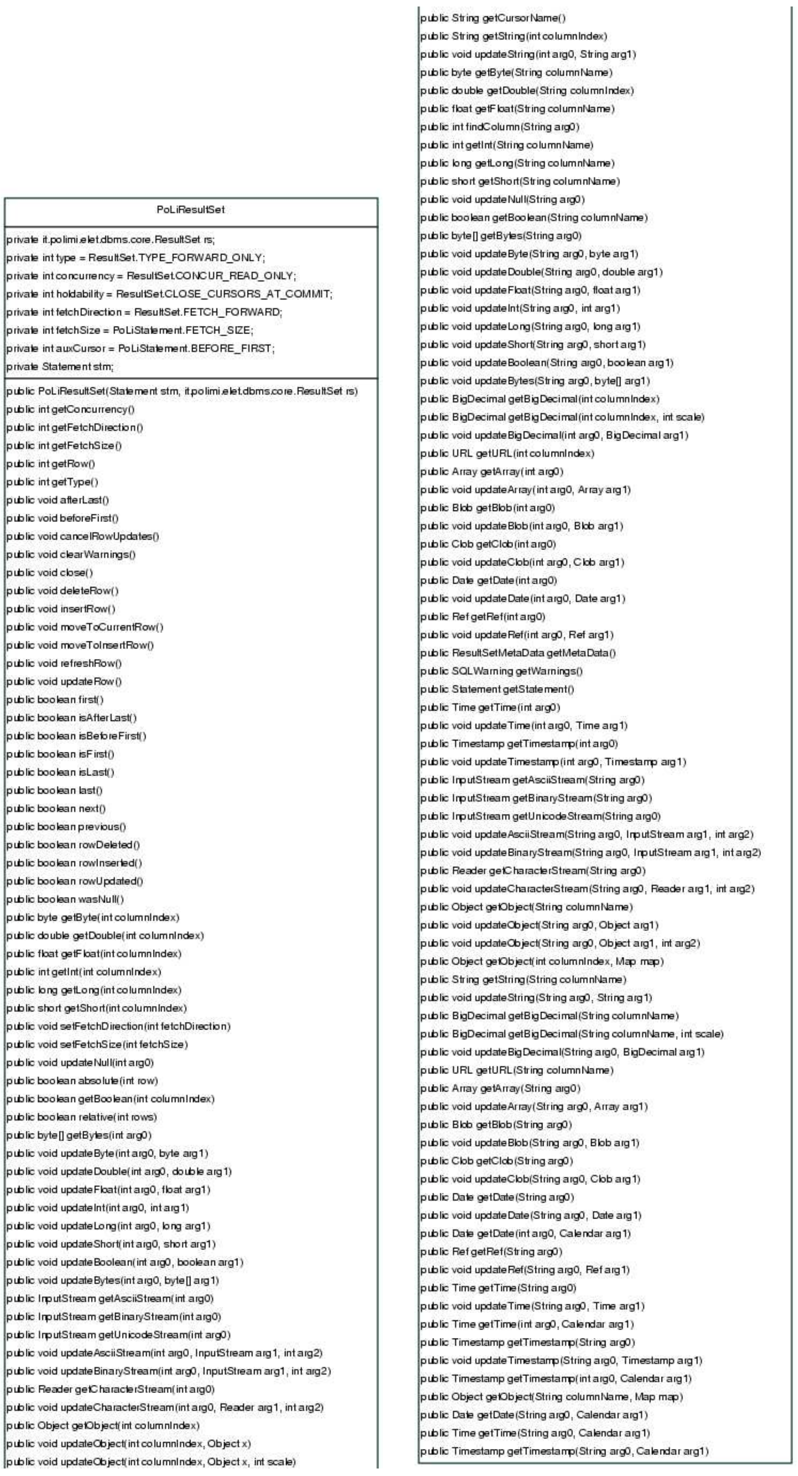

PoLiStatement
private PoLiConnection con = null;
private int type = INT_NOT_SET;
private int concurrency = INT_NOT_SET;
private int holdability = INT_NOT_SET;
private boolean readOnly = false;
private boolean autoCommit = false;
private int fetchDirection = java.sql.ResultSet.FETCH_FORWARD;
private int fetchSize = FETCH_SIZE;
private int maxFieldSize = MAX_FIELD_SIZE;
private int maxRows = MAX_ROWS;
private int queryTimeout = QUERY_TIMEOUT;
private int lastUpdateCount = 0;
private long lastOperationStarted = 0;
private long lastOperationFinished = 0;
public int getFetchDirection()
public int getFetchSize()
public int getMaxFieldSize()
public int getMaxRows()
public int getQueryTimeout()
public int getResultSetConcurrency()
public int getResultSetHoldability()
public int getResultSetType()
public int getUpdateCount()
public void cancel()
public void clearBatch()
public void clearWarnings()
public void close()
public boolean getMoreResults()
synchronized public int[] executeBatch()
public void setFetchDirection(int fetchDirection)
public void setFetchSize(int fetchSize)
public void setMaxFieldSize(int maxFieldSize)
public void setMaxRows(int maxRows)
public void setQueryTimeout(int queryTimeout)
public boolean getMoreResults(int results)
public void setEscapeProcessing(boolean enable)
synchronized public int executeUpdate(String sql)
synchronized public void addBatch(String sql)
public void setCursorName(String cursorName)
synchronized public boolean execute(String sql)
synchronized public int executeUpdate(String sql, int autoGeneratedKey)
synchronized public boolean execute(String sql, int autoGeneratedKeys)
synchronized public int executeUpdate(String sql, int[] columnIndexes)
synchronized public boolean execute(String sql, int[] arg1)
public Connection getConnection()
public ResultSet getGeneratedKeys()
public ResultSet getResultSet()
public SQLWarning getWarnings()
synchronized public int executeUpdate(String sql, String[] columnNames)
synchronized public boolean execute(String sql, String[] columnIndexes)
synchronized public ResultSet executeQuery(String query)
1
1
PoLiConnection
1
1
PoLiDataSource
1
1
<<Interface>>
DBMSServerInterface

Figure 26: A class-diagram showing how `PoLiStatement` should be implemented. Major details are related to those methods that are `synchronized`; all the execution methods wrap over the corresponding processing methods of `PoLiConnection`; in those executors a timeout counter should be implemented.



```
PoLiResultSet

private it.polimi.elet.dbms.core.ResultSet rs;
private int type = ResultSet.TYPE_FORWARD_ONLY;
private int concurrency = ResultSet.CONCUR_READ_ONLY;
private int holdability = ResultSet.CLOSE_CURSORS_AT_COMMIT;
private int fetchDirection = ResultSet.FETCH_FORWARD;
private int fetchSize = PoLiStatement.FETCH_SIZE;
private int auxCursor = PoLiStatement.BEFORE_FIRST;
private Statement stm;

public PoLiResultSet(Statement stm, it.polimi.elet.dbms.core.ResultSet rs)
public int getConcurrency()
public int getFetchDirection()
public int getFetchSize()
public int getRow()
public int getType()
public void afterLast()
public void beforeFirst()
public void cancelRowUpdates()
public void clearWarnings()
public void close()
public void deleteRow()
public void insertRow()
public void moveToCurrentRow()
public void moveToInsertRow()
public void refreshRow()
public void updateRow()
public boolean first()
public boolean isAfterLast()
public boolean isBeforeFirst()
public boolean isFirst()
public boolean isLast()
public boolean last()
public boolean next()
public boolean previous()
public boolean rowDeleted()
public boolean rowInserted()
public boolean rowUpdated()
public boolean wasNull()
public byte getByte(int columnIndex)
public double getDouble(int columnIndex)
public float getFloat(int columnIndex)
public int getInt(int columnIndex)
public long getLong(int columnIndex)
public short getShort(int columnIndex)
public void setFetchDirection(int fetchDirection)
public void setFetchSize(int fetchSize)
public void updateNull(int arg0)
public boolean absolute(int row)
public boolean getBoolean(int columnIndex)
public boolean relative(int rows)
public byte[] getBytes(int arg0)
public void updateByte(int arg0, byte arg1)
public void updateDouble(int arg0, double arg1)
public void updateFloat(int arg0, float arg1)
public void updateInt(int arg0, int arg1)
public void updateLong(int arg0, long arg1)
public void updateShort(int arg0, short arg1)
public void updateBoolean(int arg0, boolean arg1)
public void updateBytes(int arg0, byte[] arg1)
public InputStream getAsciiStream(int arg0)
public InputStream getBinaryStream(int arg0)
public InputStream getUnicodeStream(int arg0)
public void updateAsciiStream(int arg0, InputStream arg1, int arg2)
public void updateBinaryStream(int arg0, InputStream arg1, int arg2)
public Reader getCharacterStream(int arg0)
public void updateCharacterStream(int arg0, Reader arg1, int arg2)
public Object getObject(int columnIndex)
public void updateObject(int columnIndex, Object x)
public void updateObject(int columnIndex, Object x, int scale)
public String getCursorName()
public String getString(int columnIndex)
public void updateString(int arg0, String arg1)
public byte getByte(String columnName)
public double getDouble(String columnIndex)
public float getFloat(String columnName)
public int findColumn(String arg0)
public int getInt(String columnName)
public long getLong(String columnName)
public short getShort(String columnName)
public void updateNull(String arg0)
public boolean getBoolean(String columnName)
public byte[] getBytes(String arg0)
public void updateByte(String arg0, byte arg1)
public void updateDouble(String arg0, double arg1)
public void updateFloat(String arg0, float arg1)
public void updateInt(String arg0, int arg1)
public void updateLong(String arg0, long arg1)
public void updateShort(String arg0, short arg1)
public void updateBoolean(String arg0, boolean arg1)
public void updateBytes(String arg0, byte[] arg1)
public BigDecimal getBigDecimal(int columnIndex)
public BigDecimal getBigDecimal(int columnIndex, int scale)
public void updateBigDecimal(int arg0, BigDecimal arg1)
public URL getURL(int columnIndex)
public Array getArray(int arg0)
public void updateArray(int arg0, Array arg1)
public Blob getBlob(int arg0)
public void updateBlob(int arg0, Blob arg1)
public Clob getClob(int arg0)
public void updateClob(int arg0, Clob arg1)
public Date getDate(int arg0)
public void updateDate(int arg0, Date arg1)
public Ref getRef(int arg0)
public void updateRef(int arg0, Ref arg1)
public ResultSetMetaData getMetaData()
public SQLWarning getWarnings()
public Statement getStatement()
public Time getTime(int arg0)
public void updateTime(int arg0, Time arg1)
public Timestamp getTimestamp(int arg0)
public void updateTimestamp(int arg0, Timestamp arg1)
public InputStream getAsciiStream(String arg0)
public InputStream getBinaryStream(String arg0)
public InputStream getUnicodeStream(String arg0)
public void updateAsciiStream(String arg0, InputStream arg1, int arg2)
public void updateBinaryStream(String arg0, InputStream arg1, int arg2)
public Reader getCharacterStream(String arg0)
public void updateCharacterStream(String arg0, Reader arg1, int arg2)
public Object getObject(String columnName)
public void updateObject(String arg0, Object arg1)
public void updateObject(String arg0, Object arg1, int arg2)
public Object getObject(int columnIndex, Map map)
public String getString(String columnName)
public void updateString(String arg0, String arg1)
public BigDecimal getBigDecimal(String columnName)
public BigDecimal getBigDecimal(String columnName, int scale)
public void updateBigDecimal(String arg0, BigDecimal arg1)
public URL getURL(String columnName)
public Array getArray(String arg0)
public void updateArray(String arg0, Array arg1)
public Blob getBlob(String arg0)
public void updateBlob(String arg0, Blob arg1)
public Clob getClob(String arg0)
public void updateClob(String arg0, Clob arg1)
public Date getDate(String arg0)
public void updateDate(String arg0, Date arg1)
public Date getDate(int arg0, Calendar arg1)
public Ref getRef(String arg0)
public void updateRef(String arg0, Ref arg1)
public Time getTime(String arg0)
public void updateTime(String arg0, Time arg1)
public Time getTime(int arg0, Calendar arg1)
public Timestamp getTimestamp(String arg0)
public void updateTimestamp(String arg0, Timestamp arg1)
public Timestamp getTimestamp(int arg0, Calendar arg1)
public Object getObject(String columnName, Map map)
public Date getDate(String arg0, Calendar arg1)
public Time getTime(String arg0, Calendar arg1)
public Timestamp getTimestamp(String arg0, Calendar arg1)
```

Figure 27: A class-diagram showing how `PoLiResultSet` should be implemented. Note that it is internally represented by the existing result set (`it.polimi.elet.dbms.core.ResultSet`) together with the auxiliary cursor (for special positions managing). It is also has several type-oriented getters for data retrieval.

Chapter IV

JDBC™ meets PoLiDBMS

The previous chapter covered all the requirements and the specifications issues; this chapter details major features for all classes and shows what the implementation step has produced.

4.1 Implementing the main features

This section describes the output of the implementation step; significant code fragment will be used in order to better present any important feature. More precisely, the following description is intended to be an implementation report; at this point, all code has been written, let us explore now its details.

`PoLiDataSource`

Connection Tracking This point is related to the security level feature. We decided to keep track of each opened connection; this feature has been implemented through a `HashMap` and few methods (more precisely, getters and setters). The `getConnection()` (or the `getConnection(String, String)`) is in charge of creating new connections; for each created `PoLiConnection` object, a unique key is assigned and the object's reference is put into the `HashMap`, having the following tabular representation

Key	Reference
0	`it.polimi.elet.dbms.jdbc.PoLiConnection@17590db`
1	`it.polimi.elet.dbms.jdbc.PoLiConnection@14531cs`
2	`it.polimi.elet.dbms.jdbc.PoLiConnection@18522sd`
...	...

Together with the unique key, a reference to the current `PoLiDataSource` (the creator) is passed to each created `PoLiConnection`. With this feature, a two-way trusting model has been implemented, since a `PoLiConnection` object knows what its *trusted connections* are (the ones into the map) and each `PoLiConnection` object knows its *creator* (through the reference to the `PoLiDataSource` instance).
At operation request time, the `PoLiConnection` object that is requesting for a DBMS operation is checked for its validity through the following steps:

1. The key must be a valid one: its value must be less that the current stored one. This assures that the connection has been opened in the past;
2. Its reference must be into the `HashMap` in the same row of its key.

If the above test is passed then the request is served since the connection is a trusted one. The cost of this operation is declared to be constant since

1. T$(a < b) \sim \Theta(1)$
2. T(find($\$hashMap, \$reference$)) $\sim \Theta(1)$

Security Levels The above connection tracking system could be sometime unnecessary; more precisely one could desire to set different levels of trusting. For this reason, we implemented the following policy:

1. Each `PoLiConnection` object has its unique key, and this cannot be disabled;
2. Each `PoLiConnection` object knows what its related `PoLiDataSource` instance is, and this cannot be disabled;

3. When a `PoLiDataSource` object receive an operation request from a `PoLiConnection` object
    (a) *if* the security level is set to high, *then* the connection is checked for its validity;
    (b) *else* the connection is not checked for its validity and the operation is instantly performed.

So, a *high security level* corresponds to a *two-way* checking, where a connection is trusted because

1. it knows its related `PoLiDataSource` object (its creator);
2. it is known by its `PoLiDataSource` object through a quick search into the map.

On the other side, a *low security level* corresponds to a *one-way* checking. In this case, the connection is trusted only because it knows its related `PoLiDataSource` object (its creator).

Customizability Network parameters like the server port and address, the network protocol and DBMS related parameters like a brief description of the data source and the name of the database, are all customizable; this feature is implemented with one getter and one setter method for each of the above mentioned parameters.

Wrapping Methods Our implementation also provides specific `protected` methods to wrap major DBMS operation, that are:

begin — It tells the DBMS to start a new transaction; according to the set security level, an eventual connection validity check is performed, then it invokes a begin procedure directly to the transaction manager. If a transaction error occurs, an exception is thrown in order to notify this status;

commit — It tells the DBMS to commit current changes, if they exist; according to the set security level, an eventual connection validity check is performed, then it invokes a commit procedure directly to the transaction manager. If a transaction error occurs, an exception signal notify this error;

rollback — It tells the DBMS to rollback to the previous status; a connection validity check is eventually performed, according to the current security level and then the transaction manager is requested for a rollback procedure. If the rollback fails an exception will notify this, as usual;

process — It asks the DBMS to process a given (set of) operation(s). A proper connection validity check could be eventually performed and then the request is directly forwarded to the DBMS engine. If no exception occurs, this method returns the operation result: a so called `ServicePack`;

logout — It disconnects the current logged user. No validity check is performed but if something goes wrong with the logout procedure, an exception is thrown. This method does not take the connection key as an argument; it only invokes a logout procedure in general.

Except for the `logout()` one, all the above methods take the connection key as an extra argument, in order to perform the first level security check. Other versions of `logout()` method exist; they are:

`closeConnection(int)` — acts like `logout()` but it performs a connection validity check in order to ensure that the connection to be closed is a valid one;

closeConnection(PoLiConnection) — act like `closeConnection(int)` but the validity check is based upon the `PoLiConnection` reference.

Unimplemented Features Since we do not support event logging facilities, the two `setLogWriter(PrintWriter)` and `getLogWriter()` methods are not supported; if one attempts to invoke one of them, an exception signal is thrown.

Other Notes This object uses the RMI infrastructure in order to locate the (remote) DBMS through its daemon. Because of this, some exception signals are RMI related and not only DBMS related.

Because of testing purpose, additional getter methods were added in order to achieve a better introspection.

`PoLiConnection`

Statement Creation Only standard `Statement` are supported, they are realized by the `PoLiStatement` definition. Statement creators are three methods with argument overloading:

`createStatement()` — instantiates a new `PoLiStatement` object with default values for `resultSetType`, `resultSetConcurrency` and `resultSetHoldability` (these values are held by proper class fields);

`createStatement(int, int)` — instantiates a new `PoLiStatement` object with default values for `resultSetHoldability`; the other parameters are taken as argument and directly passed to the `PoLiStatement`'s constructor;

`createStatement(int, int, int)` — passes the three given argument directly to the `PoLiStatement`'s constructor and returns the statement instance.

Local Transactions Along to the standard `Connection` transaction methods – **commit** and **rollback** – two additional methods have been implemented:

begin — forwards the begin operation request to the object-associated `PoLiDataSource` object;

process — forwards the process operation request to the object-associated `PoLiDataSource` object;

They are both `protected` and the **process** method is also `synchronized`.

Unimplemented Features `PreparedStatement extends Statement` and `CallableStatement extends PreparedStatement` are not supported since the

DBMS does not support such features.
Other unimplemented features are

Warning — Warning logging is an extension of general event logging, that is not implemented;

Catalog — Since the DBMS does not have any kind of catalog, this feature together with its methods, is not implemented;

Savepoints — The transaction manager uses only one savepoint per transaction, thus this feature is not implemented.

As usual, if a method that belongs with the above features is called, an exception signal notifies that it is unsupported yet.

Other Notes The parser is not able of receiving a generic SQL statement and returning it in the DBMS supported grammar; as a result method `nativeSQL(String)` to not fully implemented: it simply return the passed `String` parameter.
A finalize method is implemented in order to invoke a connection closing procedure, letting the user not bother tho close any opened connection.
Because of testing purpose, additional getters have been written, in order to achieve the desired status introspection.

`PoLiStatement`

Constant Fields This object is also used to define and store several constants, widely used by other objects such as `PoLiResultSet` or `PoLiConnection`. There is no particular reason because this object has been chosen for this extra function; however, such constant fields follow:

`AFTER_LAST` — used by `PoLiResultSet`, it labels a cursor to be after the last position of a result set;

`BEFORE_FIRST` — used by `PoLiResultSet`, it labels a cursor to be before the first position of a result set;

`FETCH_SIZE` — used by `PoLiConnection`, `PoLiResultSet` and `PoLiStatement`, it holds the default fetching size;

`IN_BETWEEN` — used by `PoLiResultSet`, it states that a cursor is in a valid position, between the results;

`INT_NOT_SET` — globally used, it labels an integer variable to be not set yet;

`MAX_FIELD_SIZE` — used by `PoLiResultSet` and `PoLiStatement`, it stores the maximum field size in bytes;

`MAX_ROWS` — used by `PoLiResultSet` and `PoLiStatement`, it stores the maximum number of rows that a result set may contain;

`QUERY_TIMEOUT` — used by `PoLiStatement`, it holds the default timeout value for a query to be completed.

Any doubt on the usefulness of the above constants will be clarified in the following paragraphs.

Execution Methods Several (statement) execution methods, with peculiar differences and return values have been implemented. The following list describes each one of them:

`executeQuery(String)` — it creates a so called `ServicePack` and fills it with the passed query string. Then it requests the related connection for the `ServicePack` to be processed and gets back the results; these are then put into a new `PoLiResultSet`, that is returned.

`execute(String)` — it calls the `executeQuery(String)` method and states whether or not the returned object is a valid `PoLiResultSet` object. If it is a valid one it returns `true`, otherwise `false`.

`execute(String, int)` — the first argument is the SQL statement; the second one should specify whether or not the auto-generated keys have to be available for retrieval; since the DBMS does not support for auto-generated keys, this method acts *exactly* like the above `execute(String)` and the second argument is ignored.

`execute(String, int[])`, `execute(String, String[])` — the first argument is the SQL statement; the second one should specify for which columns (by index, by name) the auto-generated keys have to be available but, again, this feature is not supported for the above mentioned reason. This method acts *exactly* like the above `execute(String)` and the second argument is ignored.

`executeUpdate(String)` — it puts the given SQL statement in a so called `ServicePack` and requests its processing; then the result set size is returned. For each `Statement` object, a query execution timeout could be set; this method provides a threaded system to count time and request a rollback if the timeout expires. Such system works with two threads

Main Thread that starts the secondary thread and wait for its termination within the timeout period; if the timeout expires or if the secondary thread is interrupted a rollback procedure is tried;

Secondary Thread that is a `Runnable` object in charge of executing the requested statement;

The last method has equivalent realizations in `executeUpdate(String, int)`, `executeUpdate(String, int[])`, `executeUpdate(String,-String)`; because of the above explained reasons (the once related to auto-generated keys) these all act *exactly* like the main, that is `executeUpdate(String)`.

Shared Variables The above execution methods make large use of `Vector`, `ServicePack` and `it.polimi.elet.dbms.core.ResultSet` objects with a lot of push and pull operations; for this reason we decide to reduce the spatial and temporal complexity of such operations by creating shared class variables; their instantiation – with a relatively high cost – is performed only once and since they are like containers, a more convenient cleaning is enough before using them as new. Attention must be paid before using them because they could contain unwanted data: a test case will show that each execution method always performs the required cleaning.

Unimplemented Features In addition to the already mentioned unimplemented features, there is the warning logging facility, that is still to be unimplemented yet.

Other Notes This object also has a finalizing method that lets the user not having to close each opened statement; the garbage collector will do it automatically.

`PoLiResultSet`

Data Types With more that 60 specialized getters, this object defines which are the driver's (un)supported data types. Such getters are in the following form

`getType(int)` — that gets the specified column (by index) value as the given `Type`

`getType(String)` — that gets the specified column (by name) value as the given `Type`

Following a "polymorphism paradigm" we decided to implement only one basic getter, that is the `getObject()` (in its two forms); starting from this getter, the Java programming language is in charge of all type conversion issues, with the casting operator. For example, the `getString(int)` method has been implemented as follows

```
public String getString(int columnIndex) throws SQLException {
    try {
        return (String) this.getObject(columnIndex);
    } catch (ClassCastException e) {
        throw new SQLException("The field does not contain a string");
    }
}
```

In this way we provide a fully extensible mechanism for type conversion, completely based on the Java programming language.
The implemented data types are in the following list:

1. `OBJECT`

2. `BYTE`
3. `SHORT`
4. `INT`
5. `LONG`
6. `FLOAT`
7. `DOUBLE`
8. `BIGDECIMAL`
9. `BOOLEAN`
10. `STRING`
11. `DATE`
12. `TIME`
13. `URL`

Result Set Types We implemented two kinds of result set: *scrollable* and *non-scrollable*. JDBC™ provides two kind of scrollable result set: *sensitive* and *insensitive*; the first reads eventual changes into the underlying data source too, the second does not do that. We chosen to implement only an *insensitive scrollable* result set, together with the *non-scrollable* one; motivations are the following ones:

1. To "keep it simple"
2. For high transaction security and isolation level

These two types are fully compatible with JDBC™ `ResultSet`'s specifications.

Advanced Wrapping This object is nothing more than a wrapper around the existing `it.polimi.elet.dbms.core.ResultSet` implementation; the representation of `PoLiResultSet` is an existing result set instance.
Our implemented wrapper extends its features and adds what is specified by the JDBC™ `ResultSet` interface. This wrapper is an "advanced" one

because it has to solve several problems; the major one is the cursor position. More precisely, a JDBC™ compliant `ResultSet` must provides two particular cursor position, that are

`BEFORE_FIRST` — when the cursor is before the first result set row;

`AFTER_LAST` — when the cursor is after the last result set row.

The existing result set does not support for these positions and so we chose to *simulate* them by using two cursors; they together represents one *real* cursor.

Primary Cursor — the one provided by the existing result set;

Secondary Cursor — that assumes only three values

- `BEFORE_FIRST` — indicating that the *real* cursor is before the first row;
- `IN_BETWEEN` — indicating that the *real* cursor is the **Primary Cursor**;
- `AFTER_LAST` — indicating that the *real* cursor is after the last row.

Unimplemented Features/Data Types There is only a partially implemented feature, for which a long test session is being planned for: the update functionality, that consists in a result set that can be updatable and whose updates are directly reflected to the real data.
Along with unimplemented features there are unsupported data types; for each unimplemented type, the associate getter method throws a proper exception (telling the user that the type is not supported). More precisely, the unsupported types are complex and/or binary types, such as

1. `ASCIISTREAM`
2. `BINARYSTREAM`
3. `CHARACTERSTREAM`
4. `UNICODESTREAM`
5. `ARRAY`
6. `BLOB`

7. `CLOB`
8. `REF`
9. `TIMESTAMP`

Thanks to the adopted type supporting we left these types for future extensions.

Other Notes None.

`PoLiDatabaseMetaData`

Flexibility As mentioned in specifications, this object will be widely used to keep track of driver and DBMS development status. Major information is directly included into those getters methods and some other information is held by proper `priva-te` fields. Together with getters there are several checkers; methods for support checking (e.g. `supportTransactionLevel-(int level)`) that return a boolean value.

Status The `DatabaseMetaData` interface definition consists of more than 70 methods and about 150 constant fields; we implemented about 20 methods. It could appear strange but we must think that it was not designed for small databases and for this reason we have selected only a very little feature subset.

`PoLiResultSetMetaData`

Flexibility This object implementation status is hardly influenced by both the internal result set implementation and the `PoLiResultSet` development status; like `PoLiDatabaseMetaData` it will surely be fully implemented in the future and for this reason we always have kept it flexible (with clean code documentation and self-explaining exception throwing).

Status (unsupported features) At this time

`PoLiDaemon`

RMI Based This is a little RMI based daemon; it uses the RMI registry in order to make the DBMS available in a TCP/IP network. Since RMI works with `RemoteObject`(s) only, we modified the existing `DBMS` creating the new `DBMS-Server`, that is a `RemoteObject` that can be registered by RMI through its `DBMSServerInterface` interface.

Modularity Since running the daemon has several phases, we divided the entire daemon into some methods that are normally executed sequentially. This has been done for both code readability reasons and testing purposes. The following list describes such (`private`) methods:

`recovery()` — invokes the local transaction's status recovery procedure;

`create()` — instances the DBMS;

`register()` — creates the RMI registry and registers the DBMS instance.

They are executed by the following (`public`) methods:

`startUp()` — first invoke the `recovery()`, the `create()` and then the `register()` method;

`tearDown()` — unregisters the DBMS from the RMI registry.

Customizability This object is customizable in terms of network parameters; default values for network address and port exist but the object provides proper fields in order to let the user changes them as he/her prefers.

Status Tracking Each method outputs both normal status – on system's standard output – and error status – on system's standard error. Along with writing on the standard error, proper exceptions are thrown if an error occurs; in this way both the system administrator (at the console) and a program (by catching the exception) could be noticed of erroneous status.

4.2 Testing

As it is should be done for each software product, a JDBC™ is required to be tested for *implementation errors* and/or *conceptual errors*; the latter errors should already be avoided by the design step, but assuming the worst case is a good way to conduct testing sessions.

4.2.1 Testing Environment & Tools

Let us now illustrate our used JDBC™ testing environment: which Java Virtual Machine (JVM), which operating system, which platform and which testing applications of course. Early tests have been conducted on a x86 based PC but we have also planned to prepare an appropriate testing PDA, in order to cope with the so called *hardware limitations*.

At this time, the JVM stands over a GNU/Debian[1] Linux[2] operating system: the JVM is part of Sun®'s Java SDK 1.4.2_04 Standard Edition[3]. Since the entire development process has been supported by IBM's Eclipse[4], a fully pluggable Integrated Development Environment (IDE), this application will aid testing sessions too.

An appropriate Eclipse's facility will help us running and displaying results of several test cases; we created test cases with JUnit[5], a test-automation framework for Java applications; it helps programmers automating testing procedures by creating their own `TestCase`(s) – for class level testing – and/or `TestSuite`(s) for aggregate testing. Figure 28 shows the Eclipse IDE and the integrated JUnit interface for displaying test results. Along with an automated testing procedure, we think that "the best way to test a JDBC™ driver is to use it on a (real) application"; since we do not have a real application yet, we have chosen to implement a *dummy application*, a little application that makes use of our JDBC™ driver (please see subsection 4.2.3 for details).

[1]Available at `http://www.debian.org` or its mirrors.
[2]Linux 2.6.8.1 kernel (Aug, 23rd – 2004. Available at `http://www.kernel.org` or its mirrors.
[3]Available at `http://java.sun.com` or its mirrors.
[4]Draft Pre-release 3.0, available at `www.eclipse.org`
[5]Available at `http://junit.sf.net`

4.2.2 Class level testing

We now explain the main `TestCase`(s) by which major classes have been tested. We have tested with the following criteria

1. Does the class respect its specifications?
2. Does the class respect JDBC™ specifications?
3. Does any procedure run as fast as desired/expected?

And we have tested the following classes

```
it.polimi.elet.dbms.PoLiDaemon

it.polimi.elet.dbms.jdbc.PoLiDataSource

it.polimi.elet.dbms.jdbc.PoLiConnection

it.polimi.elet.dbms.jdbc.PoLiStatement

it.polimi.elet.dbms.jdbc.PoLiResultSet
```

Other classes – like exceptions objects – do not require any testing since they do not perform any operation/algorithm but they simply hold/display information. A brief description of each `TestCase` and its results follows[6].

`PoLiDaemonTest` ***extends*** `TestCase`

Only RMI related testing have been performed for this case. The test case is in charge of starting up the daemon with variable parameters – like different IPs and different PORTs – and then checking if

- The socket is connected (that is, it is not closed but really opened);
- The socket is bound to the correct `<IP, PORT>` couple;

Such a test case is a very simple one and we registered 100% of successful runs (see Figure 29). No more has been done for this object.

[6]All tests have been included in the `it.polimi.elet.dbms.jdbc.test` package, except for a couple of cases that have been included within the `it.polimi.elet.dbms.jdbc` package for method's visibility reasons.

`PoLiDataSource` ***extends*** `TestCase`

Since there are some `protected` methods, this test case's file is located in the `jdbc` base package for visibility reasons.

The following tests have been performed

Getting a connection to the data source — testing for `getConnection()` method's coherence; that is, it acts right in all possible object status. It always has to return a valid connection, also with multiple requests and it has to manage the connection map in a correct manner, according to the security level. More precisely, a list of performed tests follows

1. The connection key generator must be zero at object creation time;
2. A non-null `PoLiConnection` object must be returned when `getConnection()` is called;
3. After creating a new connection, the key generator must be incremented and its value must be one unit more than the one of the connection object;
4. if the security level is set to *high* $\Rightarrow$
   (a) When a connection is created, is must be putted into the connection pool with the correct key;
   (b) Whenever a connection object is closed, its reference must be removed from the connection pool;
5. if the security level is set to *low* $\Rightarrow$
   (a) When a connection is created, is must *not* be putted into the connection pool with the correct key;
   (b) Whenever a connection object is closed, the connection pool must not be modified.

Beginning/rolling-back/committing a transaction — since the transaction manager is a trusted sub-system we have tested against security permissions; that is, the *begin/rollback/commit* request is forwarded to the transaction manager if, and only if, the connection through which the request comes is a trusted one. The following list will clarify the above issues

1. if the security level is set to *high* only trusted connections' requests are really executed; this check is connection-key based;
2. if the security level is set to *low* each valid (instance of `PoLiConnection`) connection's request is executed; this check is connection-key based too.

Processing one or more statements within a transaction — testing against both security permissions and effective statement execution. The test case prepares different `ServicePack`(s) and processes them; it checks both for object/transaction integrity and `ServicePack` size and content.

Test results — The first testing session failed at all. After a standard debugging procedure, a login manager error appeared. More precisely we found that the real problem is that the login manager does not allow multiple users to execute operations; requesting more than one connection through the `getConnection()` method causes an exception to be thrown and there is no way to avoid that. We partially solved this problem both by using only one connection at time – to perform testing session – and reporting this issue to transaction manager's developers. This problem have made us thinking more carefully while creating a new transaction.

`PoLiConnection` ***extends*** `TestCase`

This test case's file is located into the proper `jdbc.test` package. Such test case does not test all methods but only the ones that performs DBMS operations or that change driver/object's status.
The following list summarizes test case's major parts:

Really close connections — Testing whether or not the `close()` methods really closes the current connection. After having opened a connection, the procedure asserts that the connection is *not* closed; thus, the closing method is invoked and the procedure asserts that the connection is closed. This test is a solid one because the connection's `isClosed()` method is not merely based upon flags or similar; it really checks if the `DataSource` still allows the current connection to request for operations.

Create valid statements — Standing on the JDBC™ API, the created `Statement`(s) are required to generate `ResultSet`(s) with certain characteristics (type, concurrency, holdability). The testing procedure ensures that those objects satisfy the API requirements; more precisely it opens a new connection and creates a new statement through which result sets are generated and checked for the following requirements:

`TYPE` — the default value must be `TYPE_FORWARD_ONLY`, unless something different is specified;

`CONCURRENCY` — the default and the only supported value is `CONCURRENCY_READ_ONLY` and no methods should be able to modify such values (e.g. setters for those values are all required to be empty and thus do nothing);

`HOLDABILITY` — the default and the only supported value is `CLOSE_CURSORS_AT_COMMIT` and no methods should be allowed to affect such attributes.

The same has been done for other attributes like `Connection`'s transaction isolation level.

Fixed TypeMap — We opted for a fixed type mapping from SQL92 to Java types instead of a customizable one; this test ensures that there are not any mechanism to modify the type mapping. More precisely every type map setter is checked to be disabled.

Valid key — This feature has already been tested within the `PoLiDataSource` testing procedure. However, few code lines have been here spent to enforce that the key assignment mechanism works right.

Test results — The above testing procedure have been passed and no particular problems have been encountered. That more than one connection is allowed to remain opened with no apparent problems, was found.

`PoLiStatement` ***extends*** `TestCase`

Like the one of `PoLiConnection`, this `TestCase`'s file is located in the proper testing package; `jdbc.test`. We tested only the main methods, that are all the *executors*.

Executors — The main statement execution method is `executeQuery(String)`; about all the other execution methods are implemented upon this one; it simply gets the requested statement as a string, executes it and returns a `PoLiReSultSet` (if the statement was a `SELECT` query, elsewhere `null` is returned). We performed several tests against this method and its variations, with both scrollable result sets and non-scrollable result sets. Such tests regard to statement *effective execution*, *correct/coherent table row insertion* and *updated row counting capability*.

Batch — A testing procedure has been written in order to test `Statement`'s capability of executing several statements (of different types), sequentially. After having added 8-10 statements through the `addBatch(String)` method and invoked the `executeBatch()`, the procedure tests that all statements are correctly executed.

Test results — A lot of failures have been encountered while testing this module. Such errors are all related to a transaction manager's strange behavior; the transaction manager is a very strict one and the automation of starting/ending a transaction seems to be quite impossible. Due to this we have chosen to modify both the `PoLiConnection` and the `PoLiStatement` object. More precisely this modification adds more control over transaction boundaries

Before modification — From outside the driver an application can control the `commit` operation only, with the `Connection#commit()` method. With the `autoCommitMode` flag, one can set whether or not a transaction has to be auto-committed;

After modification — From outside the driver an application can also control the `begin` operation, with the `Connection#begin()` method. With the `autoBegin` flag, one can set whether or not a transaction has to be auto-begun.

`PoLiResultSet` ***extends*** `TestCase`

This test's file is located in the `jdbc.test` package. It is in charge of testing various `ResultSet`'s properties and behaviors; correct cursor movement, data coherence, scrollability in different status and proper data type retrieval. The performed testing procedure can be divided into two categories and the following list explains them together with testing results

Cursor movement — There are several methods to move the cursor and check for its position; after having retrieved a 4-rows result set the following tests have been performed (a scrollable result set is assumed)

`absolute(int)` — `absolute(1)` must be the same as `first()` and `absolute(-1)` must be the same as `last()`. Other operation with negative integer must be coherent with the following table

Positive Position	Negative Position
1	-4
2	-3
3	-2
4	-1
...	...

The above property has been checked with two `for` cycles that move-and-check the cursor at every iteration.

`relative(int)` — the simplest way to check this method is that it must satisfy the following property

$$\texttt{relative(rows)} \iff \texttt{absolute(cursorPosition() + rows)}$$

where the `cursorPosition()` function returns the current cursor position into the result set. After having successfully tested against the `absolute(int)` method, the above property guarantee that the `relative(int)` method works too.

`first()` — After having invoked this method, the procedure asserts that the cursor is on the $1-st$ row and that the `isFirst()` method returns true.

`last()` — After having invoked this method, the procedure asserts that the cursor is on the $n-th$ row and that the `isLast()` method returns true.

`beforeFirst()` — After having invoked this method, the procedure performs two checks

1. trying to retrieve a row causes an exception signal to be thrown;
2. the method `isbeforeFirst()` returns true.

`afterLast()` — After having invoked this method, the procedure performs two checks

1. trying to retrieve a row causes an exception signal to be thrown;
2. the method `isAfterLast()` returns true.

`getRow()` — This method has does not have a *real* testing procedure; it is used all around the test case and we also trust on it because it is just a wrapper of the internal `ResultSet#getCursorPosition()` that has been already tested.

`isFirst()` — This method has to check whether or not the cursor position is the $1-st$; the testing procedure moves the cursor on the $1-st$ position by using different methods and then checks if this method returns true when other checkers do so.

`isLast()` — This method has to check whether or not the cursor position is the $n-th$; the testing procedure moves the cursor on the $n-th$ position by using different methods and then checks if this method returns true when other checkers do so.

`isBeforeFirst()` — Since there is only one way to move the cursor after the last row, we have to trust on the `beforeFirst()` and use it to move the cursor and then check for `isBeforeFirst()` to return true.

`isAfterLast()` — Since there is only one way to move the cursor after the last row, we have to trust on the `afterLast()` and use it to move the cursor and then check for `isAfterLast()` to return true-

`next()`, `previous()` — The testing procedure runs two `for` cycles that traverse the entire result set in the two opposite directions (from the first to the last and from the last to the first); during such cycles some assertions about data coherence and cursor position are performed.

Types retrieval — At this time we have been able to insert only integer data into DBMS' tables; for this reason no tests have been performed against other type getting methods. We only checked the `getObject(int)` (`getObject(String)`) and the `getInt(int)` (`getInt(String)`) methods. The written testing procedure traverses a result set and retrieves the column values on which simple value-assertions are performed.

Test results — Through this test case, a lot of cursor movement's bugs were found; the main cursor movement's methods have been corrected because they were not coherent between each others. In addiction to this, some methods have been found to be JDBC™ uncompliant. Such errors have been all fixed.

4.2.3 Testing through a running example

Since JDBC™ is an API, it is always used between (real) applications and (real) data sources; for this reason the best way to test a JDBC™ driver is stressing it through a (real) simple application; for example, a graphical interface to the DBMS' core or a text-based console.
We created both a graphical console and text-based console; let us now illustrate how they work and how they are used for testing purpose.

`ui.JDBClient`

It is the real DBMS' client; it makes use of the JDBC™ driver in order to get a connection to the data source and abstracts the query submission procedure within a method called `submit(String)`. Thanks to this method, an application can instantiate a new `Client` object and simply use it to send queries[7].

Its constructor instantiates objects of the following classes: `DataSource`, `Connection` and `Statement`. Through these objects, `submit(String)` forwards the

[7]Note that a query is intended to be a `SELECT` statement.

query to `Statement`'s `executeQuery(String)` and returns a new `ResultSet` object. Note that the DBMS' data source is empty; the constructor is also in charge of filling some tables with simple integer records. The following code fragment explains what is mentioned above

```
...
public ResultSet submit(String query) {
    try {
        return this.stm.executeQuery(query);
    } catch (SQLException e) {
        return null;
    }
}
...
```

Note that `null` is returned if a failure occurs.

`ui.Textual`

This object uses a static `JDBClient` reference within its `main(String [])` in order to submit queries from the command line – `System.in` – and print results to `System.out` in tabular form. An example follows

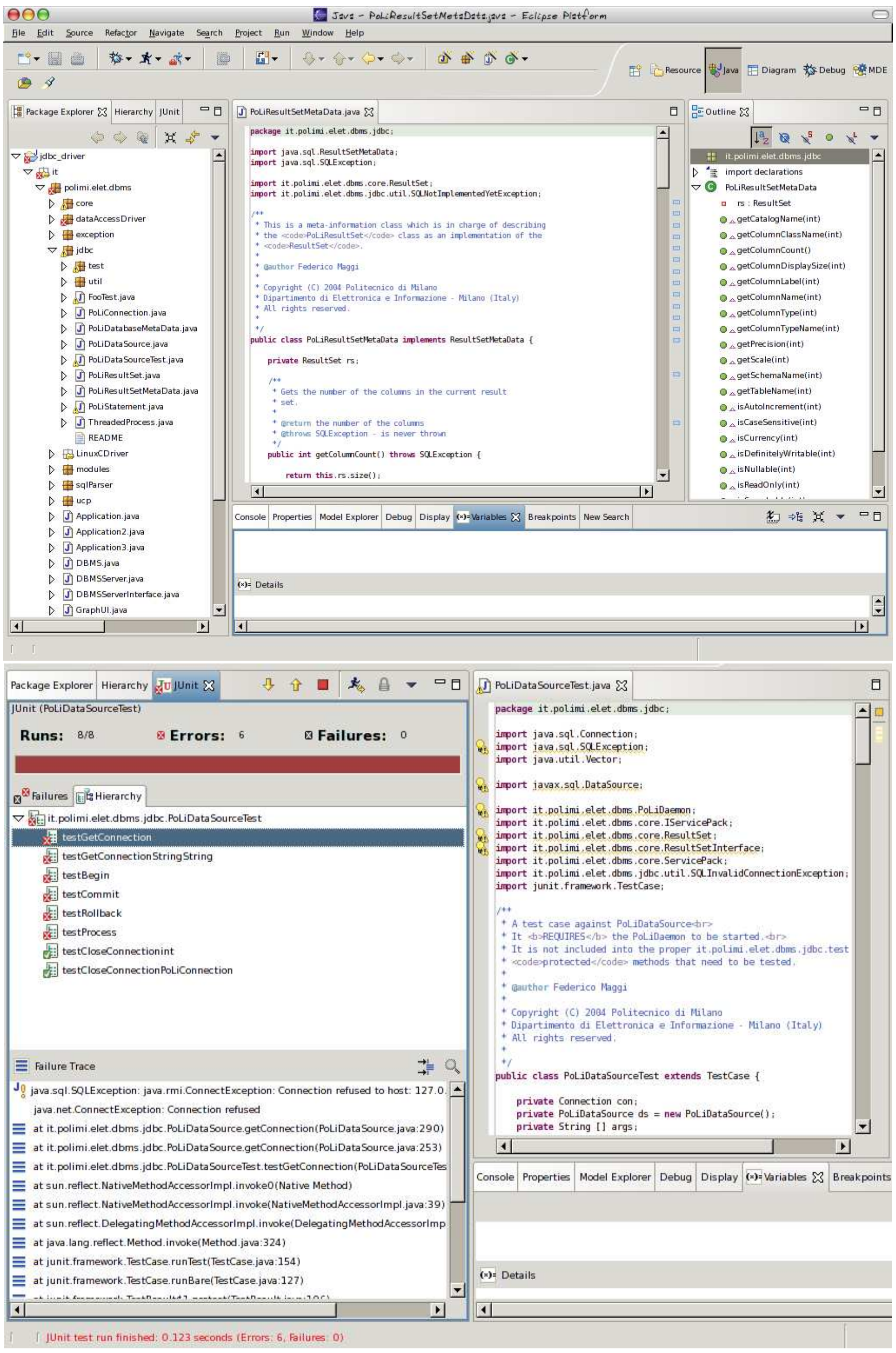
```
...
select * from tab1
1, 1, 2
1, 3, 1
2, 1, 3
1, 2, 2
1, 1, 3
...
```

We used this interface to send a lot of testing queries; as a result, an error in `PoLiResultSetMetaData#getColumnCount()` has been found and fixed.

`ui.Graphical`

This object is the graphical equivalent of `uc.Textual`; consisting of an input text field and an output text area, it displays query results whenever a new (valid) query is submitted by pressing a button. Figure 30 reports a screenshot of `Graphical` on work.

On the following section (A.1), how these two interfaces use JDBC™ is better explained.



PoLiResultSetMetaData.java

```
package it.polimi.elet.dbms.jdbc;

import java.sql.ResultSetMetaData;
import java.sql.SQLException;

import it.polimi.elet.dbms.core.ResultSet;
import it.polimi.elet.dbms.jdbc.util.SQLNotImplementedYetException;

/**
 * This is a meta-information class which is in charge of describing
 * the <code>PoLiResultSet</code> class as an implementation of the
 * <code>ResultSet</code>.
 *
 * @author Federico Maggi
 *
 * Copyright (C) 2004 Politecnico di Milano
 * Dipartimento di Elettronica e Informazione - Milano (Italy)
 * All rights reserved.
 *
 */
public class PoLiResultSetMetaData implements ResultSetMetaData {

    private ResultSet rs;

    /**
     * Gets the number of the columns in the current result
     * set.
     *
     * @return the number of the columns
     * @throws SQLException - is never thrown
     */
    public int getColumnCount() throws SQLException {

        return this.rs.size();
```

JUnit (PoLiDataSourceTest)

Runs: 8/8 Errors: 6 Failures: 0

```
it.polimi.elet.dbms.jdbc.PoLiDataSourceTest
    testGetConnection
    testGetConnectionStringString
    testBegin
    testCommit
    testRollback
    testProcess
    testCloseConnectionint
    testCloseConnectionPoLiConnection
```

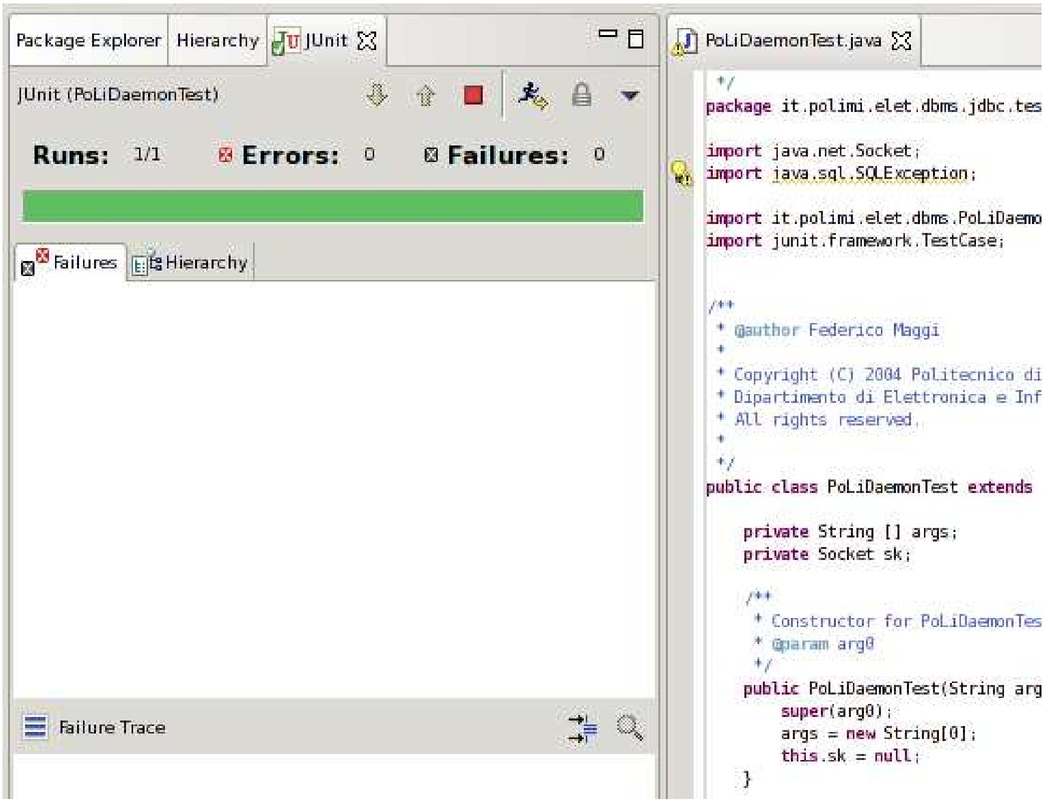
```
java.sql.SQLException: java.rmi.ConnectException: Connection refused to host: 127.0.
 java.net.ConnectException: Connection refused
 at it.polimi.elet.dbms.jdbc.PoLiDataSource.getConnection(PoLiDataSource.java:290)
 at it.polimi.elet.dbms.jdbc.PoLiDataSource.getConnection(PoLiDataSource.java:253)
 at it.polimi.elet.dbms.jdbc.PoLiDataSourceTest.testGetConnection(PoLiDataSourceTes
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp
 at java.lang.reflect.Method.invoke(Method.java:324)
 at junit.framework.TestCase.runTest(TestCase.java:154)
 at junit.framework.TestCase.runBare(TestCase.java:127)
```

JUnit test run finished: 0.123 seconds (Errors: 6, Failures: 0)

PoLiDataSourceTest.java

```
package it.polimi.elet.dbms.jdbc;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.Vector;

import javax.sql.DataSource;

import it.polimi.elet.dbms.PoLiDaemon;
import it.polimi.elet.dbms.core.IServicePack;
import it.polimi.elet.dbms.core.ResultSet;
import it.polimi.elet.dbms.core.ResultSetInterface;
import it.polimi.elet.dbms.core.ServicePack;
import it.polimi.elet.dbms.jdbc.util.SQLInvalidConnectionException;
import junit.framework.TestCase;

/**
 * A test case against PoLiDataSource<br>
 * It <b>REQUIRES</b> the PoLiDaemon to be started.<br>
 * It is not included into the proper it.polimi.elet.dbms.jdbc.test
 * <code>protected</code> methods that need to be tested.
 *
 * @author Federico Maggi
 *
 * Copyright (C) 2004 Politecnico di Milano
 * Dipartimento di Elettronica e Informazione - Milano (Italy)
 * All rights reserved.
 *
 */
public class PoLiDataSourceTest extends TestCase {

    private Connection con;
    private PoLiDataSource ds = new PoLiDataSource();
    private String [] args;
```

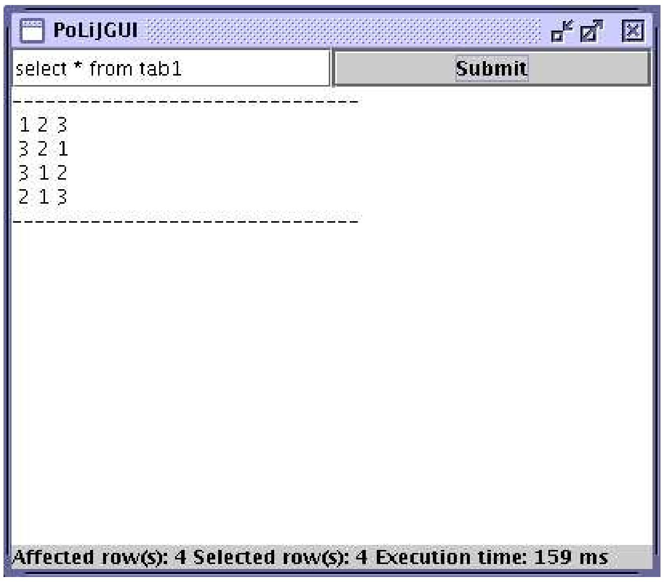
Figure 28: Two screenshots showing (top) how Eclipse appears and (bottom) how Eclipse integrates JUnit.



Runs: 1/1 Errors: 0 Failures: 0

```
 */
package it.polimi.elet.dbms.jdbc.tes

import java.net.Socket;
import java.sql.SQLException;

import it.polimi.elet.dbms.PoLiDaemo
import junit.framework.TestCase;


/**
 * @author Federico Maggi
 *
 * Copyright (C) 2004 Politecnico di
 * Dipartimento di Elettronica e Inf
 * All rights reserved.
 *
 */
public class PoLiDaemonTest extends

    private String [] args;
    private Socket sk;

     /**
      * Constructor for PoLiDaemonTes
      * @param arg0
      */
    public PoLiDaemonTest(String arg
        super(arg0);
        args = new String[0];
        this.sk = null;
    }
```

Figure 29: Testing against `PoLiDaemon`.



```
select * from tab1
```

```
------------------------------
 1 2 3
 3 2 1
 3 1 2
 2 1 3
------------------------------
```

Affected row(s): 4 Selected row(s): 4 Execution time: 159 ms

Figure 30: The GUI of our running example.

Chapter V

Conclusions and Future Development

The JDBC™ technology API specification is 190 pages long; the entire API is made up of two packages and about 30 interfaces for 15948 lines of code. JDBC™ has been designed to support virtually any kind of data source and it is able to offer an API for 'huge' DataBase Management Systems like Oracle. On the other side, a palm computer has a limited memory and CPU equipment and a not so long power resource durability. Such a limited resources scenario, together with PoLiDBMS, a custom dbms for Very Small Databases, are the elements that led to the definition of a special driver to access the stored data.

Let us now briefly analyze the achieved result. The initial project phase has focused on the analysis of the JDBC specification, overviewing the entire API,examining existing drivers, exploiting PoLiDBMS and understanding its transaction manager's policies. This preliminary step has given us required information about the technological environment. Along with those detailed concepts, a first idea of the driver architecture started to be defined.

The definition of the *scaling down procedure* workflow, has taken a relatively brief period. The existent methodology ([6]) had made us familiar with such kind of approach. Since we have already acquired a precise idea of the entire API, finding the main working points was an immediate task.

After having briefly reviewed the produced workflow, we tested it during the requirement analysis and the design specification step; here, we had the opportunity to *see our ideas at work*.

Classical requirements' analysis and design specifications steps have been carried out with care for any single interface to be designed. Any object/interface has been analyzed from different point of view; **What** is its role? **Where** does it stay? **How** does it perform its task? In this way, all possible details have been addressed for a more precise design step. This accurate design procedure resulted in a *fast* and *error-less* implementation step.

For additional/final testing sessions we reckon a running example, a testing application designed to be both a *test case* and a *code sample* for the user manual.

At this time, PoLiJDBC is not a fully JDBC™ compliant driver because of both PoLiDBMS' constraints and the required limited size. Anyway, several improvements have been taken into account so the future development will be as easy as possible.

Even if a driver prototype has been released, this project still under development as well as PoLiDBMS. Considering new incoming PoLiDBMS modifications, we are planning of performing some *refining activities* and adding new driver advanced functionalities or completing unimplemented features (see the paragraph **Advanced features** in Section 2.1).

Distributed Transactions Management According to the JDBC™ `XADataSource` extension, adding supports for distributed transactions is strongly required since PoLiDBMS is meant to be used in a distributed environment, as Figure 9 shows. The development of such a feature could reckon the existing distributed transaction manager facility w.r.t. the JDBC™ framework. Offering a standard API for distributed transactions will made PoLiDBMS a more complete-but-small system.

Updatable Result Sets With the incoming release of the new XML Data Access Driver, the existing transaction manager can reckon a more solid base for commitment operations. Thanks to such a new enhancement, the `ResultSet` unimplemented methods could be safely added and instantiating different `Statement`s for `UPDATE`s will be no longer required.

Metadata Information As we mentioned in Section 4, `PoLiDatabaseMetaData` should be used to track the development status of both this driver and PoLiDBMS. Adding new functionalities or modifying existing ones will results in a more detailed/complete `DatabaseMetaData` implementation; in the same way the CVS repository tracks the PoLiJDBC history, `PoLiDatabaseMetaData` should be used as a useful *changelog* for future works.

Appendix A

DOCUMENTATION AND USER MANUAL

The user manual is here included with some sample code. This chapter does not include a developer reference, reported in the next chapter.

A.1 End user manual

For the user manual we reckon the JDBC™ 3.0 API Documentation and other tutorials by Sun®, covering the standard use of a hypothetical fully implemented API. Because we implemented only the needed API subset, here we explain all driver-specific aspects and peculiarities (for advanced users) with some examples of use and a quick *"getting started"* paragraph. All examples are build with respect to our *running example* (see 4.2.3 for an introduction).

A.1.1 Getting started

This the end user manual; it has been written in a tutorial form to make it well accessible and more direct for a generic user. That the user has an already installed PoLiDBMS copy, is assumed; we start from the JDBC™ driver deployment and then we propose an example of use, that is our running example.

Installing/deploying the driver To get the driver working properly, a minimal network environment is needed; we assume that the machine has the default loopback device that is addressed with the universal `localhost`'s IP: `127.0.0.1`. The binary release of the JDBC™ driver has the following package tree:

```
it.polimi.elet.dbms.

    DBMSServerInterface.class
    DBMSServer.class
    PoLiDaemon.class

    jdbc.
        PoLiDataSource.class
```

```
PoLiConnection.class
PoLiStatement.class
PoLiResultSet.class
PoLiResultSetMetaData.class
PoLiDatabaseMetaData.class
util.
   SQLInvalidConnectionException.class
   SQLNotImplementedYetException.class
   SQLNullPointerException.class
   SQLUnsupportedTypeException.class
   SQLWrongTypeException.class
   SQLWrongValueException.class
```

We deployed it into a Java ARchive, JAR, compressed archive so it is only required to copy it into PoLiDBMS' root path. For example, if PoLiDBMS is installed under

```
/usr/local/polidbms/
```

the correct location for `polijdbc.jar` is the same path so their package trees match.
If the JVM is told to run the `polijdbc.jar` package, the `it.polimi.elet.dbms.-PoLiDaemon.main(String [])` is started by default and the daemon listens for incoming connections from `PoLiDataSource` on `127.0.0.1:2005`. If a different `<IP, PORT>` couple is needed they can be specified as follows

```
[user@polisrv]$ java -jar polijdbc.jar 151.99.123.2 2005
```

and the daemon will be started accordingly. A port scan on the machine is strongly recommended in order to see if the specified port is busy.

```
[root@polisrv]\# nmap localhost

Starting nmap 3.55 ( http://www.insecure.org/nmap/ ) at 2004-09-09 17:07 CEST
Interesting ports on localhost (127.0.0.1):
(The 1656 ports scanned but not shown below are in state: closed)
PORT     STATE SERVICE
22/tcp   open  ssh
2005/tcp open  deslogin
```

If no errors/exceptions occur the daemon is properly started up; let us now illustrate how to implement a simple application that makes use of PoLiDBMS through PoLiJDBC, the driver.

Example of use We are going to implement a little JDBC™ based client for PoLiDBMS; as shown in section 4.2.3, that client will be used to create two different console: a graphical one and a textual one. Figure 31 better explains how our running example application is logically organized into different layers.

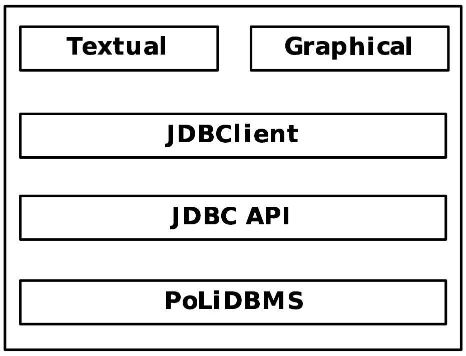


Figure 31: The logical structure of the running example application.

In order to abstract the query submission mechanism from a console down to PoLiDBMS, we pass through `JDBClient` and the JDBC™ API. The simplest way to submit a query and retrieve its results is to get a connection to `DataSource`, through `Connection` interface, and create a `Statement` object to submit queries. Figure 32 shows the class diagram of `JDBClient`.
The constructor instantiates a new `DataSource` and invokes the `private init()` method which performs major initializations steps; they are shown in the following code fragment

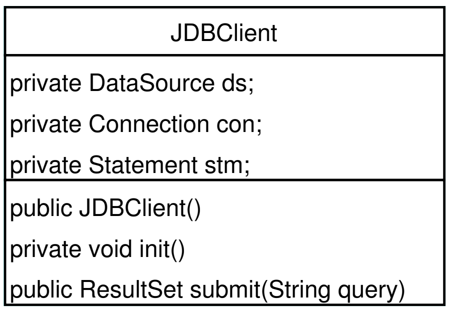


Figure 32: `JDBClient`'s class diagram.

```
private void init() {

  try {
    //Get the connection
    this.con = this.ds.getConnection();

    //Set transaction boundaries auto-execution
    this.con.setAutoCommit(false);

    //Extra API method; downcast required
    (PoLiConnection)this.con).setAutoBegin(false);

    //Create a new statement
    this.stm = this.con.createStatement(
                     ResultSet.TYPE_SCROLL_SENSITIVE,
                     ResultSet.CONCUR_READ_ONLY,
                     ResultSet.CLOSE_CURSORS_AT_COMMIT
            );

    //Fill some tables
    (PoLiConnection)this.con).begin();
    stm.executeQuery("insert into tab1 values(1,2,3)");
    stm.executeQuery("insert into tab1 values(3,2,1)");
    stm.executeQuery("insert into tab2 values(1,2,3)");
    stm.executeQuery("insert into tab2 values(3,2,1)");
    //this.con.commit();

  } catch (SQLException e) {

    e.printStackTrace();
    System.exit(1);
  }
}
```

Now we use the above declared Statement object to abstract the query submission mechanism; we can do it in the following way

```
public ResultSet submit(String query) {

  try {
    return this.stm.executeQuery(query);
  } catch (SQLException e) {
    return null;
  }
}
```

Statement#executeQuery(String) will perform all required operations on the underlying data source and a ResultSet implementation object or null will be returned.

Now we can implement the two boundaries interfaces; Textual and Graphical. They both use the JDBClient#submit(String) method to send submitted queries and retrieve their results; more precisely, the following iteration is used

```
String line = reader.readLine().trim().toLowerCase();
...
ResultSet rs = client.submit(line.trim());
String results = "";
try {
  int columns = rs.getMetaData().getColumnCount();

  while (rs.next()) {
    for (int i = 0; i < columns; i++)
      results += rs.getInt(i) + ", ";

    results += "\n";
  }

  System.out.println(results);
} catch (SQLException e) {
  e.printStackTrace();
  System.exit(1);
}
```

First, the query – `line`'s value – is submitted to the client and the returned result set is saved. The columns count is retrieved through a meta information object; the first `while` cycle iterates over the entire result set and the nested `for` cycle extracts one field per cycle. The last operation prints results on the standard output. Note that `Graphical` and `Textual` differs only on the last operation; while `Textual` prints on `System.out`, `Graphical` prints on a Java Swing `TextArea`.

The following sections deal with the use of basic functionalities of JDBC™, focusing on our particular implementation.

A.1.2 Basic functionalities

Let us recall the three main purposes of a JDBC™ driver, they were first reported on section 2.1; our explanation will follow the same order; that are:

1. **Establish a connection with a DBMS**;
2. **Send SQL statements to the data source**;
3. **Retrieve and process the results**.

A detailed description of each step is reported below.

How to establish a connection? — First, the daemon must be started and binded to the desired `<host, port>`; we remember that this operation need a previously installed PoLiDBMS release and a PoLiJDBC JAR. We invoke the JVM with the following line

```
[user@host]$ java -jar polijdbc.jar 127.0.0.1 1057
```

that automatically starts the daemon.

Second, the application that requires to get a connection must use a `DataSource` implementation, that is `PoLiDataSource`. The following code fragment details how to establish a custom connection

```
/* step 1 */
DataSource ds = new PoLiDataSource();
((PoLiDataSource)ds).setServerName("127.0.0.1");
((PoLiDataSource)ds).setPortNumber(1057);

Connection con = ds.getConnection();
((PoLiConnection)con).setAutoBegin(false);
con.setAutoCommit(false);

((PoLiConnection)con).begin(); //required
```

Note that some down-casts are required to access some extra API features. With an established connection one can create multiple statements and ask the DBMS for different operations; let us to illustrate how to do this.

How to send SQL statements to the data source? — After have followed the above step, to create a `Statement` object is required. An application should operate as follows, in order to create a statement:

```
/* step 2 */
//Establish a connection (step 1)
Statement stm = con.createStatement(
                  ResultSet.TYPE_SCROLL_SENSITIVE,
                  ResultSet.CONCUR_READ_ONLY,
                  ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

The above code specifies that all returned results set will be scrollable; other flags are ignored while unsupported yet. If needed, one can get multiple statements from the same connection. The application can now send SQL statements through `Statement` objects; the simplest method to do that follows:

```
/* step 3 */
//Establish a connection (step 1)
//Create a statement (step 2)

stm.executeQuery("insert into tab1 values (1,2,3)");
stm.executeQuery("insert into tab1 values (2,2,1)");
ResultSet rs1 = stm.executeQuery("select * from tab1");
ResultSet rs2 = stm.executeQuery("select tizio from tab1");

con.commit();
```

After having inserted a couple of records, the procedure collects the desired data into two `ResultSet` objects. According to the specified `ResultSet` type, one can now retrieve rows and columns in the same ways that JDBC™ API explains.

This brief tutorial has explained how to exploit basic functionalities; advanced features are disclosed in the following.

A.1.3 Advanced functionalities

Now we will investigate the main advanced functionalities; they are *extra API features*, *exceptions* and *JDBC™ advanced operations*.

PoLiJDBC has two main extra API features; the first is on `PoLiDataSource` which differs from what `DataSource` declares:

{`get, set`}`ServerName` — allows to retrieve or change the default TCP/IP address on which the daemon is listening; the default value for this attribute is `''127.0.0.1''` or `''localhost''`.

{`get, set`}`PortNumber` — allows to retrieve or change the default TCP/IP port number on which the daemon is listening; the default value for this attribute is 2005.

{`get, set`}`OnlyTrustedConnections` — allows to retrieve or change whether or not only trusted connections are allowed to send commands to the DBMS;

the default value for this attribute is `false`.

`{get, set}Description` — allows to get or submit the description of the data source; the default value for this attribute is `‘‘Hi!  It is PoLiDBMS.’’`.

`getCon{Key, Map}` — allows to retrieve the current connection key or the current connection map; the default values for these two attributes are `-1` and `null`, respectively.

`get{Username, Password}` — not used but added for possible future requirements; the related attributes both have `null` as default value.

Another extra API feature is owned by `PoLiConnection`; this object have two extra methods, that are:

`setAutoBegin` — sets whether or not the transaction manager has to start a new transaction automatically, that is whenever a new query is submitted. This attribute is `true` by default but we strongly recommend to set it `true`.

`begin` — tells the transaction manager to start a new transaction; this is often used to achieve manual control of transaction boundaries. We strongly recommend to use this method after having disabled the so called *auto-begin mode*.

Special Exceptions subclasses Along with the above improvements there is an advanced exception tree; as we mentioned in specifications' section (see 3.2.3), we designed this particular tree in order to achieve a more detailed error reporting. The generic `SQLException` is dispatched into several sub types with different meanings:

`SQLInvalidConnectionException` — reports that a `Connection` instance is not a valid one.

`SQLNotImplementedYetExcepion` — is thrown whenever a code fragment attempts to access an unsupported feature. It could be thrown by any method of any class.

`SQLNullPointerException` — has the same meaning of `NullPointerException` but it is intended to be used in JDBC™ environment.

`SQLUsupportedTypeException` — indicates that a certain (SQL) type is not supported. It could be thrown by any `getType` method of `PoLiResultSet`.

`SQLWrongValueException` — reports that a certain attributes has not the expected value. It could be thrown by any method of any class.

A.1.4 Sample code

This section provides a list of sample code for common operations like *batch statements*, *result retrieval*, *transaction commitment* and more. This section is not meant to be complete or covering all PoLiJDBC features; instead, to give some significant examples is its main purpose.

Note — `PoLiDaemon` must be running before executing the following code samples. Calling `java -jar polijdbc.jar` is the fastest way to to start up the DBMS daemon (see A.1.1). If the daemon is properly started, each code sample can be taken and executed *as a whole*.

Filling a table with new rows — This sample shows how to set up all required objects to connect to PoLiDBMS through PoLiJDBC and insert a couple of records into `tab1`. First, a new `PoLiDataSource` object is instantiated; from this object, a new `PoLiConnection` is created with the appropriate method. After having set up some transaction related parameters, a new transaction is started and a new statement is created. Thanks to `executeUpdate(String)` two `INSERT` statements are sent to PoLiDBMS.

```
package it.polimi.elet.dbms.jdbc.samples;

import it.polimi.elet.dbms.jdbc.PoLiConnection;
import it.polimi.elet.dbms.jdbc.PoLiDataSource;

import javax.sql.DataSource;

import java.sql.Statement;
import java.sql.Connection;

public class TableFiller {

        public static void main(String[] args)
                throws Throwable {

                //initializates the main objects
                DataSource ds = new PoLiDataSource();
                Connection con = ds.getConnection();
                ((PoLiConnection)con).setAutoBegin(false);
                con.setAutoCommit(false);
                Statement stm = con.createStatement();

                //begin
                ((PoLiConnection)con).begin();

                //adds some rows
                stm.executeUpdate("insert into tab1 values(1,2,3)");
                stm.executeUpdate("insert into tab2 values(3,2,1)");

                //commit
                con.commit();
        }
}
```

Retrieving results — This sample shows how to send a `SELECT` statement through PoLiJDBC and to retrieve results into a `ResultSet` realization. First, a new `PoLiDataSource` object is instantiated; from this object, a new `PoLiConnection` is created with the appropriate method. After having set up some transaction related parameters, a new transaction is started and a new statement is created. Note that through this new statement we specify that all returned `ResultSets` are required to be *scrollable*. `executeQuery(String)` is used in order to send the query and to get its results.
First, the result set is scrolled *from the first to the last* with the use of `next()`; then, after having moved the cursor *after* the last row, a new `while` cycle iterates over the result set *from the last to the first* row. The generic `getObject(int)` method have been used to retrieve each column value.

```
package it.polimi.elet.dbms.jdbc.samples;

import it.polimi.elet.dbms.jdbc.PoLiConnection;
import it.polimi.elet.dbms.jdbc.PoLiDataSource;

import javax.sql.DataSource;

import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.Connection;

public class TableRetriever {
        public static void main(String[] args)
        throws Throwable {

                //initializates the main objects
                DataSource ds = new PoLiDataSource();
                Connection con = ds.getConnection();
                ((PoLiConnection)con).setAutoBegin(false);
                con.setAutoCommit(false);
                Statement stm = con.createStatement(
                                ResultSet.TYPE_SCROLL_INSENSITIVE,
                                ResultSet.CONCUR_READ_ONLY,
                                ResultSet.CLOSE_CURSORS_AT_COMMIT);

                //begin
                ((PoLiConnection)con).begin();

                ResultSet rs = stm.executeQuery(
                                    "select * from tab1");
                int columns = rs.getMetaData().getColumnCount();

                //first to last
                while (rs.next())
                        for (int i = 0; i < columns; i++)
                                System.out.println(
                                    rs.getObject(i) + ", ");

                //last to first
                rs.afterLast();
                while (rs.previous())
                        for (int i = 0; i < columns; i++)
                                System.out.println(
                                    rs.getObject(i) + ", ");

                //commit
                con.commit();
        }
}
```

Batch of statements — This sample shows how to create a batch of different statements by using the `addBatch(String)` method. First, a new `PoLiDataSource` object is instantiated; from this object, a new `PoLiConnection` is created with the appropriate method. After having set up some transaction related parameters, a new transaction is started and a new statement is created. Through this new statement, the procedure adds a sequence of statements to the batch; by calling the `executeBatch()` method, the statements are executed from the first to the last and a `int []` is returned in order to describe how many rows have been modified per statement.

```
package it.polimi.elet.dbms.jdbc.samples;

import it.polimi.elet.dbms.jdbc.PoLiConnection;
import it.polimi.elet.dbms.jdbc.PoLiDataSource;

import javax.sql.DataSource;

import java.sql.Statement;
import java.sql.Connection;

public class Batch {
        public static void main(String[] args)
                throws Throwable {

                //initializates the main objects
        DataSource ds = new PoLiDataSource();
        Connection con = ds.getConnection();
        ((PoLiConnection)con).setAutoBegin(false);
        con.setAutoCommit(false);
        Statement stm = con.createStatement();

        //begin
        ((PoLiConnection)con).begin();

        //adds some statements to the batch
        stm.addBatch("insert into tab1 values(1,2,3)");
        stm.addBatch("insert into tab2 values(3,2,1)");
        stm.addBatch("update table tab1 set tizio=22");
        stm.addBatch(
        "update table tab1 set caio=33 where tizio=22");

        //execute the batch
        stm.executeBatch(); //also clears the batch

        //commit
        con.commit();
        }
}
```

Metadata information — This sample shows how to retrieve and explore the main meta information for both a `DataSource` and a `ResultSet`. First, a new `PoLiDataSource` object is instantiated; from this object, a new `PoLiConnection` is created with the appropriate method. After having set up some transaction related parameters, a new transaction is started and a new statement is created. A new query is sent to PoLiDBMS through the new statement. Then, the procedure starts to retrieve several meta information about the returned `ResultSet`: *how many fields does the result contains? Is a certain column a read only one? What is the name of a certain column?*. The last code fragment retrieves information about both the driver and the underlying data source.

```
package it.polimi.elet.dbms.jdbc.samples;

import it.polimi.elet.dbms.jdbc.PoLiConnection;
import it.polimi.elet.dbms.jdbc.PoLiDataSource;

import javax.sql.DataSource;

import java.sql.Statement;
import java.sql.Connection;

public class Batch {
        public static void main(String[] args)
                throws Throwable {

                //initializates the main objects
        DataSource ds = new PoLiDataSource();
        Connection con = ds.getConnection();
        ((PoLiConnection)con).setAutoBegin(false);
        con.setAutoCommit(false);
        Statement stm = con.createStatement();

        //begin
        ((PoLiConnection)con).begin();

        //adds some statements to the batch
        stm.addBatch("insert into tab1 values(1,2,3)");
        stm.addBatch("insert into tab2 values(3,2,1)");
        stm.addBatch("update table tab1 set tizio=22");
        stm.addBatch(
        "update table tab1 set caio=33 where tizio=22");

        //execute the batch
        stm.executeBatch(); //also clears the batch

        //commit
        con.commit();
        }
}
```

Appendix B

DEVELOPERS MANUAL

Because of this different nature, the developers manual has been released in a separate publication and into two different formats; Hyper Text Markup Language format, for fast browsing, and paper format (see [10]).

Appendix C

GLOSSARY

On this section, a brief PoLiJDBC reference is reported; it has not to be taken as a complete manual since it is only a small glossary.

API — PoLiJDBC implements a subset of the JDBC™ 3.0 API. A list of the implemented classes follows: `DataSource`, `Connection`, `Statement`, `ResultSet`, `ResultSetMetaData`, `DatabaseMetaData`. Section 2.1 provides an overview of the API; sections 2.2 and 2.2.1 provides a more detailed explanation of our implementation.

Batch Statements — PoLiJDBC let an application to send batch SQL statements to the DBMS to be executed sequentially. As JDBC™ API specify – see 2.1.1 (**Advanced Features**) – each `PoLiStatement` object can accumulate a certain number of SQL statements and send them at 'one command'.

Client — An example of JDBC™ client has been implemented upon PoLiJDBC; it can be found at `it.polimi.elet.dbms.jdbc.ui` together with a brief documentation (see 4.2.3).

Close Statements/Connections — To close each opened `PoLiConnection` and `PoLiStatement` object is strongly recommended; our implementation provides 'finalizing' methods in order to automatically close those objects when they are no longer referenced.

Daemon — A network daemon for PoLiDBMS is implemented by `PoLiDaemon`. Please see **Network Protocol** for major details.

Data Access — This driver uses the newest `DataSource` approach with an hybrid approach; please see section 2.2.1 for more details.

Distributed Transactions — An implementation of the distributed transaction functionality has been planned for future development. PoLiDBMS already has an high-level distributed transaction manager that could be programmatically accessed. For more details see [1].

Driver Name — This driver's name is **PoLiJDBC**; with the same acronym used to name PoLiDBMS, this name stands for **Po**rtable **Li**ttle **JDBC** (driver). The prefix **PoLi** also indicates that the driver has been designed to work with PoLiDBMS.

Error Reporting — PoLiJDBC includes an advanced error reporting infrastructure; it makes use of an exception tree in order to provide a very detailed and highly semantic error reporting. See sections A.1.3 and 3.2.3 (`SQLException` paragraph) for details.

Metadata — Metadata interfaces – `ResutlSetMetaData` and `DatabaseMetaData` – are not fully implemented by `PoLiResutlSetMetaData` and `PoLiDatabaseMetaData`, respectively. The latter object is mainly used to track the development status of both PoLiJDBC and PoLiDBMS.

Multithreading — All methods that could be accessed concurrently are `synchronized`; they are: `PoLiDataSource#getConnection()`, `PoLiDataSource#getConnection(String, String)` and all `PoLiStatement`'s execution methods. PoLiDBMS' transaction manager and core are both multithread safe, so PoLiJDBC over PoLiDBMS could be concurrently used.

Network Protocol — This driver works over the TCP/IP protocol. The daemon, `PoLiDaemon`, is reachable at `127.0.0.1:2005` by default.

Result Set — PoLiJDBC provides the `PoLiResultSet` object to store queries results. Since, it is not fully JDBC™ 3.0 compliant, to read code documentation is strongly recommended.

Transaction Boundaries — Standard JDBC™ requires a `commit()` method to be implemented in order to commit a transaction. PoLiDBMS' transaction manager needs a `begin()` method to be invoked before any transaction, so `PoLiConnection` has an extra method, `begin()`, to perform major control over transaction boundaries. Together with the above feature, another extra method has been added; `PoLiConnection#setAutoBegin(boolean)`.

Please see section A.1.3 for major details about these advanced functionalities.

Types — PoLiJDBC supports a subset of SQL92's types according to what PoLiDBMS supports. A list of the supported types follows: `OBJECT`, `BYTE`, `SHORT`, `INT`, `LONG`, `FLOAT`, `DOUBLE`, `BIGDECIMAL`,
`BOOLEAN`, `STRING`, `DATE`, `TIME`, `URL`.

REFERENCES

[1] A. Lazaric, S. Sheffer, C. Pascali, "Local and distribuited transaction manager (source code documentation)," tech. rep., 2003.

[2] A. Miele, "The design and prototype development of a database management system for portable devices," Master's thesis, Politecnico di Milano - Dipartimento di Elettronica e Informazione, 2003.

[3] Alonso, Korth, "Database system issues in nomadic computing," pp. 388–392, 1993.

[4] Barbara Liskov with John Guttag, *Program developement in Java - Abstraction, Specification and Object-Oriented Design.* Addison Wesley, Jan 2000.

[5] C. Bobineau, L. Bouganim, P. Pucheral and P. Valduriez, "PicoDBMS: Scaling down database techniques for smart card," in *26th International Conference on Very Large Databases*, pp. 11–20, 2000.

[6] C. Bolchini, F. A. Schreiber, L. Tanca, "A methodology for very small data base design," *preprint submitted to Information Systems*, 2004.

[7] C. Bolchini, F. Salice, F. A. Schreiber, L. Tanca, "Logical and physical design issues for smart card databases," *ACM Transactions on Information Systems, Vol. 21*, July 2003.

[8] C. Curino, M. Giorgetta, "The design and prototype development of a database management system for portable devices," Master's thesis, Politecnico di Milano - Dipartimento di Elettronica e Informazione, 2003.

[9] E. Ansuini et al, C. Batini, B. Pernici, G. Santucci, F. Angeli, *Sistemi Distribuiti.* 2001. serie Sistemi Informativi, vol. V.

[10] F. Maggi, "Polijdbc developers manual (source code documentation)," tech. rep., 2004.

[11] Forman, Zahorjan, "The challenges of mobile computing," vol. 17 (4), pp. 38–47, 1994.

[12] Jim Conallen, *Building Web applications with UML.* Addison Wesley, 2002.

[13] Jon Ellis & Linda Ho with Maydene Fisher, "Jdbc™ 3.0 specification - final relase," tech. rep., Oct 2001.

[14] L. Galluppi, F. Maggi, G. Scorta, "Database portabili: un caso di studio.," tech. rep., 2004.

[15] M. Fisher, J.Ellis, J. Bruce, *JDBC™ API Tutorial and Reference, Third Edition.* Sun® Microsystems, Inc., May 2003.

[16] Sun® Microsystems, Inc., *For Driver Writers, [15, p. 1097–1119]*, 2002. avaiable online at `http://java.sun.com/products/jdbc/driverdevs.html`.

[17] Zaslawsky, Tari, “Mobile computing: overview and current status,” pp. 1–8, 2001.